FUZE®

Get with the PROGRAM

- Understand computing
- Learn to program
- Discover electronics
- Debug and fix your code
- Explore robotics
- Prepare for your future

Project workbook
for FUZE BASIC

Written by Jon Silvera
& Colin Bodley

About the FUZE

The FUZE started out as a personal project intended to teach my own children how computing looked back in my day... the 1980s.

When you turned on a computer back then it wouldn't load Windows, Mac OS X or Android, etc., but just quietly and very quickly boot to a text based screen usually with a >Ready prompt and a little flashing cursor.

To get the computer to do things you typed in direct commands like **PRINT** and **LET A=10**, or listings of commands called 'programs'.

And so it began...

Jon Silvera
Founder

f u z e • c o • u k

FUZE ®

Please Note

The FUZE Project Workbook is a work in progress.

Please check the FUZE website under resources for new projects, tutorials and lesson plans.

FUZE Projects should be considered complimentary to the curriculum, not all encompassing.

FUZE BASIC offers an ideal introduction to text based ('real') programming and as such works very well alongside graphical environments like Scratch, Kodu, Espresso and more complex text based ones like Python, Java and C.

With un-quantifiable gratitude to Shila, Gracie, Molly, David, teamFUZE, Geoff Graham, Gordon Henderson, MikeDX and John Ronane (bless im)

& equal thanks to Atari, Acorn, Commodore, Sinclair, Amstrad, Microsoft and Apple.

& Douglas Adams and Isaac Asimov

Index

F U Z E ®

GET WITH THE PROGRAM

Getting started
"Lighting the FUZE"

In this project you will learn;

Setting up the FUZE computer system
What is a computer?
Computer basics
About computer memory

For educators:

Mapping with the Curriculum

An introduction to the core components making up a computer system.

Connecting, switching on and loading the operating system. Selecting and executing applications.

A simple introduction to computer memory and how information is digitally stored both in RAM and storage memory.

Written by Jon Silvera

About your FUZE

The FUZE is a unique computing platform. It has been designed to aid users learn and develop computing skills. It comes with everything you need right out-of-the-box. So before we get started let's just make sure you have everything and know how to set it up.

In the box
First off is the FUZE itself. You can't miss it, it looks like this...
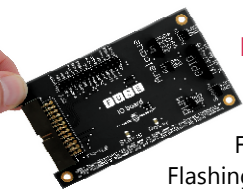
FUZE IO board
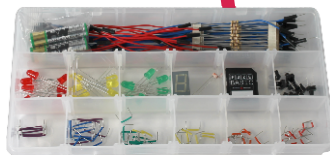The FUZE IO board is where you connect the FUZE to the real world. Flashing lights, sensors, robotics and much, much more await...

Motherboard
The computer inside the FUZE. This little board contains the CPU (**C**entral **P**rocessing **U**nit), RAM (**R**andom **A**ccess **M**emory), storage memory, USB sockets and a whole lot more, which we will look at later.

Electronic components kit
The project kit includes 24 x coloured LEDs, 1 x seven segment LED, 1 x light dependent resistor, 8 x micro switches, 30 x resistors (mixed specifications), 20 x Jumper cables, 60 x Jumper wires, mouse batteries and an SD card to boot your FUZE.
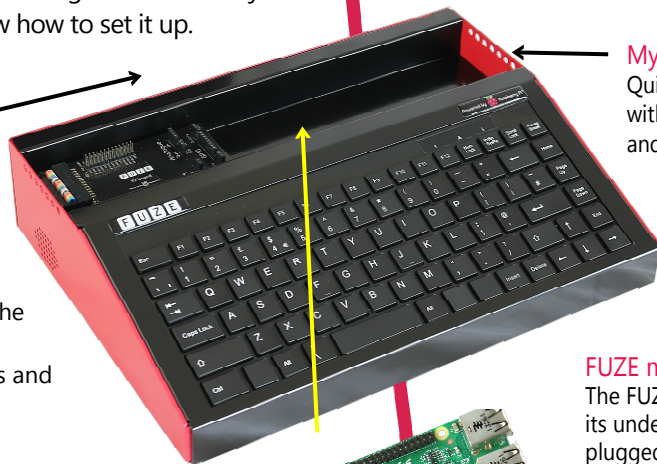
My FUZE has got holes in it!
Quite right too. The holes are compatible with many popular plastic building blocks and electronic construction kits.

FUZE mouse & mat
The FUZE mouse has a WiFi dongle (plug) in its underside. This needs to be removed and plugged in to one of the USB ports on the back of the FUZE. Also two AAA batteries are required. These are included in the electronic components kit. Please note it also has a small on/off switch underneath.

A bread board..... excuse me, did I hear you correctly?
This is used to build simple electronic circuits. In the olden days, actual wooden bread boards were often used for building prototype circuits and the name just stuck.

Setting up your FUZE

1. Connect a display, the power supply and mouse
2. Insert the SD Card (face down & carefully). Ensure the display is on and then switch on the FUZE

Only power on the system after you have inserted the SD Card. The system will not boot otherwise.

Power Socket
Plug the FUZE power supply to the wall socket and then the small end into the FUZE here. Please make sure the FUZE power switch is off when connecting.

On / Off switch
It's best to shutdown the operating system from the Desktop before switching off. Only switch off once the small red light on the FUZE IO board has gone out.

Four available USB ports
Connect various USB devices to these ports including your wireless mouse dongle (supplied) and robot arm kit (optional). You can connect USB flash drives to easily transfer programs and information.

Audio Jack
If you're not using a HDMI display with built in speakers then you can connect separate speakers to this socket. You need to run the Raspberry Pi config program to tell the FUZE to send audio out via this socket.

HDMI Display
Connect your HDMI TV or monitor here. If you have an older VGA or DVI display you will need an adapter (available from the FUZE website).

Internet / Network socket
To connect to the Internet or a network via a hub connect the network cable here. Otherwise you can plug a Wi-Fi dongle into a USB port.

SD / MicroSD slot
This slot is where you insert the FUZE SD card. This is the equivalent to a computer's hard disk and as such should be handled with care. Do not remove or insert unless the red light on the FUZE IO board is off.

Everything you were afraid to know about computers but had to ask!

Before we dive into the main projects why not take in a page of ~~useless~~ really useful information...

So what is a computer? "That's easy" I hear you say, "a computer is a magic box that makes our world better because it has games, the Internet, YouTube and Google". Well yes, but how on earth does it do these things, what's going on inside the magic box?

I'll try and make this quick...

A computer is an electronically powered calculating machine. Even an iPad / phone / Playstation / XBOX / Nintendo etc., are all just overgrown calculators. Although another way of looking at it would be to say a computer is an electronic human... don't believe me?, read on...

A computer has a number of main parts which allow us to make use of them. It's hard to list this in an order of importance as each part is pretty much as important as the rest but here goes;

The CPU or Central Processing Unit. It is easiest to think of this as a computer's brain. In fact it works in very similar ways to our own brain.

Memory, again similar to humans, is used to store information, and in computer terms, *everything* is stored in memory. The picture on an iPad screen, the sound from a film, the music you download, every single YouTube video, the key presses you type in an email or search engine, the movements from a mouse or on a touch screen and so on, everything is stored in computer memory.

Computers have two kinds of memory, quick access, which is available only when it has power (or is alive!) and storage memory which is recorded to a permanent storage device like a hard drive, a DVD / Blue-Ray disc, USB flash drive or as is popular nowadays, in the Cloud!

GET WITH THE PROGRAM
Getting started
"Lighting the FUZE"

Humans also have two kinds of memory - internal and external. Internal is what we use all of the time to remember and store information. External can be compared with encyclopaedias, dictionaries, the Internet, phones, books, invoices, comics, story books, documentaries, films, photos and even music notation.

Next up are the input and output devices. Humans have senses like sight, hearing, touch, taste and nerves for heat or pain. Computers have keyboards, mice, touch screens, monitors, cameras and electronic sensors for heat, light, motion, location and so on.

The difference between computers and humans today is intelligence. Whereas we humans have had a few hundred thousand years to evolve our 'programs', computers have only just arrived. By comparison their 'programs' are just being born and they are evolving very quickly or at least, we are evolving them.

Lastly, a computer needs a source of power like a battery or a plug in a wall socket. Computers cannot convert food into energy just yet!

Yes but how do they actually work?

Let's go back to the brain, sorry, the computer's CPU. The CPU is quite simply, millions of tiny switches called transistors and logic gates. Depending how they are set up the computer can be programmed to do different things. Information is sent to the CPU, then the CPU performs an operation on that information and then sends the result to whereever it needs to go.

Imagine a picture made from a grid ten little dots long by ten tall. Each dot can be any colour. With only a 10 x 10 grid we couldn't draw anything detailed but it can give us an idea.
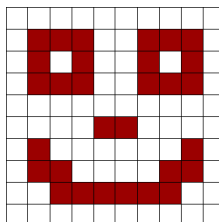
Now imagine that the very bottom left hand corner is position number one and the very top right hand corner is position one hundred. This gives us 100 memory locations we can store information.

If we use these memory locations for our display then we can control what is displayed just by changing the values of each location.

For example if dark red has a value of 10 and bright red is 100 then we could increase the brightness of all the dark red pixels in our image just by adding 90 to them. To do this a program is required. The program is also stored in memory somewhere and could look something like this;

Memory = 1 // Setup our memory counter to start at position 1
LOOP UNTIL Memory > 100 // Starts a loop that will stop at 100
Value = PEEK Memory // reads a memory location and saves it in the variable Value
IF Value = 10 THEN Value = Value + 90 // if it is dark red add 90
POKE Memory, Value // write the new value back to memory
Memory = Memory + 1 // increase the Memory counter
REPEAT // go back to the start of the loop

So our new image would now look like this;

With only ten by ten pixels we can't do anything serious but I'd like to try and get things in context so bear with me...
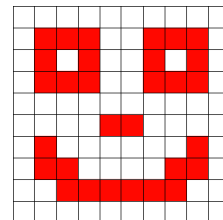
To make each pixel a different colour takes a single unit of memory and each unit of memory (known as a **BYTE**) can only hold a number up to 255 (actually 256 but computers always start from zero). So this means a picture can only have 256 different colours right... no, wrong!
What if we joined more units (Bytes) together, in fact how about we use three bytes for every pixel.

Light behaves differently to paint in that we use Red, Green and Blue light to make up any colour. Computers use each byte to store a maximum of 256 shades of each colour component, so the first byte is a shade of red, the second is green and the third is blue. This means each pixel is made up from three bytes each with a maximum of 256 shades of R, G or B.

How many colours per pixel do we have now? well the sum is 256 x 256 x 256 = 16,777,216. 16 million colours for each and every pixel.

The picture on the right is just 20 pixels wide by 30 tall. Even with such a small picture it starts to look like a photograph.

Ok but what's this got to do with computer memory?

Well this picture is 20 pixels wide x 30 pixels tall and each pixel is using three bytes of memory. To store this image therefore takes 20 x 30 x 3 bytes of memory. That's 1,800 bytes to store this tiny little image.

In the early days of computing memory was incredibly expensive to make so computers came with very small amounts of it.

The very first home computers shipped with as little as 1K of memory, that's 1,000 bytes. Actually it is 1,024 bytes because computers don't generally think in decimal or base 10 as it is known, they think in base 2, or binary. More on that later. The K in this case means kilo or kilobyte.

1 **kilobyte** requires 1,024 memory locations, 1 **megabyte** is 1,024 x 1024 **bytes** or 1 million memory locations and a **gigabyte** is 1024 **megabytes**, a **terabyte** and then a **petabyte** follow.

Early computers could not have even displayed the above image let alone store it in their tiny memory. Computers in the nineteen eighties were very limited indeed.

A modern digital camera today can take pictures with a pixel resolution as high as 5,000 x 5,000. Work out how many bytes that needs;

Remember, 5,000 x 5,000 x 3 (for the pixel colour), that's around 73 megabytes of memory for a single picture!

* Do you recognise the incredibly famous handsome man in the picture? Whad'ya mean "no".... It's me you fool, me! You'll get marks deducted for that, you'll see, they'll all see..

Memory is still expensive, although not like it was early on. However we would need tons of the stuff if computers today stored all our photos, movies, music and of course homework in their main memory.

This is why computers also need storage memory and why we talk about hard disks. We use hard disks to store information so it can be accessed whenever we need it.

A computer's main memory is called **RAM**, short for **R**andom **A**ccess **M**emory. This is the computer's working memory and, just like your brain, if you switch it off, the information it has within is lost forever. Very sad indeed.

Storage memory is therefore just as important as RAM as it allows us to keep information forever.

When we talk about **Cloud** storage it couldn't actually be further from the truth. **Cloud** storage is quite simply very powerful computers with gigantic amounts of storage memory, connected to the Internet. They are often stored underground to keep them protected. Hmm, the Cloud eh...!

We'll come back to how computers work later on, for now let's get programming...

9

FUZE®

Project stage 1-1
"Hello World"

In this project you will learn;

To write a simple program including;
A Loop
Displaying text
Text colours
Delaying program flow

Commands introduced
LOOP, REPEAT, INK, PAPER, WAIT()
and PRINT

Written by Jon Silvera

For educators:
Mapping with the Curriculum

Although an incredibly simple program, 'Hello World' introduces basic program flow with loops and delays.

The main principles are displaying text on screen, a simple loop and adding text colours.

Students are encouraged to change the text, delays and colours.

Key Stages 1, 2 & 3

Part 1 - The basics

This project shows you how to write a simple program. We're using the easiest programming language of them all, **BASIC.**

A computer program is a bit like giving someone directions. Imagine if we wrote down the directions and handed them to whoever was asking;

"Hmm... the toy shop you ask, well, you need to..."

```
Go straight ahead until you get to Apple Lane
Turn left into Toshiba Street
Straight ahead until you see Android Village Hall
Turn right into Samsung Boulevard
Straight ahead until you reach Google Towers
Straight over into Microsoft City Centre
The toy shop is the second store on the right
```

This is very similar to a computer program. A simple list of instructions telling the user (or a computer) what to do. The whole thing is called the **Program** and the instructions are often referred to as an **Algorithm**.

Let's get started..

All programmers, all of them, every single one ever, started out with this program "Hello World"*

Start up your FUZE as explained on page 5 and once on the FUZE Desktop

look for an Icon on the screen like this **FUZE BASIC** and double click it to begin.

You will see a black window with a big **FUZE** logo at the top **[Pic 1]**.

Type in; **Hello World** - and then press **Enter**.
The **Enter** key has an arrow like this

You will get an error as shown in **[Pic 2]**. This is because there's no **PROGRAM** in the computer's memory. Don't worry, it's nothing serious, as you will soon see.

* This of course is completely untrue.

B A S I C ?

Beginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode

Why oh why does it have to sound so complicated. It simply means it is a language made from easy to understand instructions that can be used for just about any kind of program.
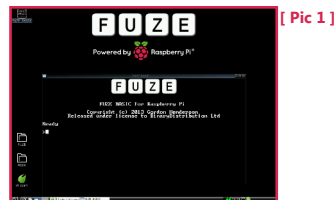
[Pic 1]

[Pic 2]

PROGRAM - a list of instructions or commands that tell a computer what to do, when to do it and what colour to do it in. Also known as Software.

All computers, mobile phones, pads, tablets, game consoles, electronic toys, washing machines, televisions and even cars use software or programs to make them work.

Learning to program is like learning to control the world...

Mwaaah hahaahaha!

Press **F2 ***. Now we're getting somewhere **[Pic 3]**. This is the Program Editor. It allows us to enter and edit programs. If there is already a program there press the **F12** key. This will erase any previous program. Now, using the editor, type in;

Editor

```
Cycle
Print "Hello World"
Repeat
```

Before we **RUN *** this program we should save our work. See the **IMPORTANT** note on the right of this page.

Let's take a closer look to see if we can work out what our program is going to do. Take each line, one-at-a-time;

Cycle – Hmm.. That's not much to go on - try the next line;

Print "Hello World"

This one makes more sense. It's going to print the words "Hello World", but wait, we don't have a printer! That's Ok because **Print** in this case means, print to the screen display.

And so to the last line; **Repeat**

What does it all mean? Well, the first line of our program says **Cycle** and the last line says **Repeat.** The **Cycle** statement tells the program this is the beginning of a loop. The **Repeat** statement says go back to the **Cycle** statement and '**repeat**' anything in between.

Keep going, we're on the right track

* F1, F2, F3 The Function keys

Across the top of your keyboard you will find a line of keys beginning with an **F** and numbered from 1 to 12. These are called the **Function** keys. They can be programmed to do specific things. Press **F1** in **the FUZE BASIC** editor to see a list of Function keys and what they do.   **[Pic 3]**
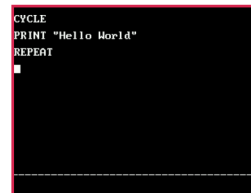
* RUN The **RUN** command simply tells the computer to execute or start the program that is in memory. When you turn on an iPad for example, the device RUNs the software to make it do all of those lovely things we take for granted.

SAVE - IMPORTANT - SAVE

It's very important to **save** our work, as when a computer is shut down, any unsaved work is lost! To **save**, press **F5** when in the editor (or **F3** to **save** & **RUN**). Delete any previous filename and then enter your own, for example, "Jon hello" (without the quotations) and then press **Enter**. It is best to avoid using any special characters in file names, so stick to letters and numbers to be on the safe side.

What do you think will happen when we **RUN** the program?

Take a little pause to think about it . . . . . . Let's try it and see if you were right. Press **F3** to **RUN** the program.

Whoa! How do we stop it? **[Pic 4]** Press the **Esc** key. This will stop the program from running. In future we will say press **Esc** or even just **exit** the program. **[Pic 5]**

Press **F2** to return to the program editor.

So what happened? The last line, `Repeat`, tells the computer to go back to the first line, `Cycle.`

So repeat it did, and then it Printed "Hello World" again, then to the Repeat statement again, and so it did, back to Cycle, Print "Hello World" and Repeat again, and so it did, back to…. Ok, I think you get the idea!

Let's try something else. In the Editor, move the cursor (the white square in the editor) to just after the last " and add a semi colon **;**

Editor

```
Cycle
Print "Hello World";
Repeat
```

I wonder if anyone actually reads this bit…
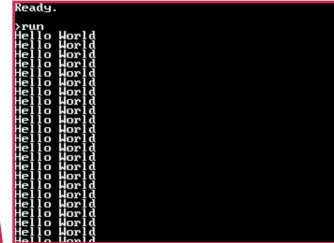
[Pic 4]

Memory
A computer generally has two types of memory. The first is its working memory where it keeps everything running when it is switched on. The second is where everything is stored when it is turned off.
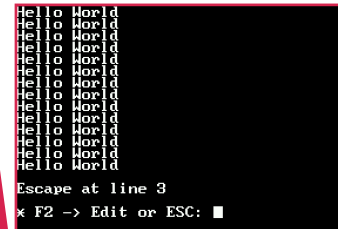
[Pic 5]

If you switch a computer off then everything in the working memory is lost. We save or store programs in storage memory so we can use them whenever we want without having to type them in all over again.

Note: To **RUN** the program within the **Editor** press **F3**. To Exit the **Editor** press the **Esc** key. You can then **RUN** a program by typing in **RUN** and pressing **Enter**. The **Editor** has its own set of keyboard commands that we will learn about later. If you are not in the **Editor** for any reason then on the main screen type **"edit"** and press the **Enter** key to get back to it.

It's not much but it will make quite a difference. Can you see, we've added a Semi Colon **;** just after the quotation (") marks.

This tiny addition tells the computer to print the next item following the last one on the same line, and not on the next line down. Press **F3** to **RUN** the program **[Pic 6]**

Yuck! The words are all joined up so it's hard to read, and it's too fast. Press **Esc** and then **F2** to return to the Editor. We can put this right in no time! **[Pic 7]**

Move the cursor to just after the "d" and insert a space - *just press the space bar*.

Then move the cursor to the end of the same line and press **Enter**.

This will insert a new line. Now add a new command; **Wait(1)**. So now you should have;

Editor
```
Cycle
Print "Hello World ";
Wait(1)
Repeat
```

Press **F3** to exit and **RUN** the program. If everything's gone to plan then your program should display "Hello World" at a nice steady pace and should have spaces in all the right places.

I wish I was a Viking, strong bold and striking…

Make no mistake…
A computer never gets it wrong. A computer PROGRAM never gets it wrong. The only mistake a computer *can* make is when a programmer has written an error in the program. The computer simply does what it is told to do and nothing more. Try and remember this when you're shouting at the computer - it's not its fault!

[Pic 6]

[Pic 7]

As you can see here, even missing out a single space in our program makes a difference. The computer doesn't understand we have spaces between words so we have to program them in.

Ok we have some explaining to do. We added a new command, `Wait(1)`. `Wait(1)` is actually pretty straightforward, especially if you watch the program running.

`Wait(1)` tells the computer to wait for a specified time. In this case, for 1 second. If you wanted to use less than a second you need to specify a number with a decimal point so 0.5 is half a second or 0.1, a tenth of a second.

It's time to brighten things up a bit. All this black and white text is really dull, don't you think?

Pick a colour, any colour…. Actually not any colour, pick one of these **[Pic 8]**;

However, please do not pick black or else we'll end up in no-end of trouble, as black text on a black screen makes things a bit difficult to read!

This picture has next week's winning lottery numbers on it, but as it's black text on a black background we can't read it. Oh well, we live and learn!

Important Note: When programming, it's important to keep your listing clean and tidy. Using capital letters at the beginning of words helps to make things much easier to read. Press **Esc** and then **F2** to return to the Editor.

Did you spot the deliberate colour mistake?

Computing and colour

Computers can generally display many more colours than those shown below. This is just an easy reference for text colours.

Actually computers usually work with only three different colours, **R**ed, **G**reen and **B**lue or **RGB**. Each colour has 256 shades from very dark to very light. By mixing these together a total of 16,777,216 (256 x 256 x 256) colours can be displayed which is lucky as this is about as many as the human eye can manage!

And another thing, computers do not mix colours like paint. Computers display colour with light. Mixing light gives different colours to mixing paint. For example, **blue light** and **green light** makes **cyan**, **red light** and **green light** makes **yellow**. **Red**, **green** and **blue** light makes **white** or **grey** if you only use a bit of each.

0	1	2	3
Black	Navy	Green	Teal
4	5	6	7
Maroon	Purple	Olive	Silver
8	9	10	11
Grey	Blue	Lime	Aqua
12	13	14	15
Red	Pink	Yellow	White

[Pic 8]

Now add the following line to your program; not the blue bit!

Editor	(remember… F2 to return to the Editor)

```
Cycle
Ink=Lime  or the colour you picked
Print "Hello World ";
Wait(1)
Repeat
```

Press **F3** to **RUN** the program. Isn't that pretty…

Press **Esc** to exit the program and **Esc** again. We don't want to go back to the Editor this time as we are going to enter a few commands directly.

Now we have a new problem. Everything's gone green. Well everything's changed to the colour you picked. This might be Ok, but if you chose something very bright or very dark then it will make it hard to read and edit the program. Before we go back to the editor type in **Ink=white** and press **Enter**. This will set the colour to white again.

Ink in **FUZE BASIC** sets the text colour, and **Paper** sets the background colour.

You can try this out before going back to the editor. Set the colours to something you like! **[Pic 9]**.

Hey, stop looking out of the window!

Colour fun!

When you are not in the **FUZE BASIC** editor you can type in and run single commands directly. Try **CLS** for example. This means clear the screen! Remember, you need to press **Enter** after each command.

Now you can try different colour combinations. Try these;

```
Ink=Red
Paper=White
CLS
```

Wow!

If you get stuck, use **CLS** to clear the screen and then enter;

```
Paper=black
Ink=white
CLS
```

This will return us to the normal settings. Phew!

[Pic 9]

Now go back to the editor (**F2**) and edit your program so it looks like this;

Editor

```
Cycle
Ink=Pink
Paper=Blue
Print "Hello World ";
Wait(1)
Repeat
```

Press **F3** to **RUN** the program. **[Pic 10]**

How does that look?

Press **Esc** to exit the program, **F2** to return to the editor and then try a few different combinations. You'll also see that the colour settings are now the same for the **Editor**.

IMPORTANT
Finally for this project you need to **SAVE** your program. If we don't **SAVE** it then when you switch off the computer the program is lost forever. Press **F5** to save the program.

[Pic 10]

Programming is for boys, not girls...
RUBBISH! Did you know the first ever actual computer program was written by a young lady called Ada Lovelace. Ada is famous for inventing computer programming and there is even a programming language named after her, "ADA". Ada worked with another very important inventor, Charles Babbage who invented the first ever programmable computer called the Analytical Engine - you can see why they got on so well!

Stage 1-1 Assessments

Now we've written our first program let's see what we've picked up so far.

Assessment 1 ⭐

Can you change the program to display 'Hello Your Name' and not 'Hello World' as before? I hope you put your actual name and not the words, 'Your Name'!

Assessment 2 ⭐

How could you make the program go really fast or really, really slowly? What was the command now... let's just **Wait** a moment and think about it!

Assessment 3 ⭐

What about different colours? Try and make the program display different colours. Hmm. "**Ink=**"

Assessment 4 ⭐⭐⭐

Ok this one is a bit harder. Could you add another line of code to display (Print) some more text, "My name is Pi" for example. What about in another colour too and hey, while you're at it, add another **Wait** to keep it all running smoothly!

End of Stage 1-1

Wow, this coding lark is fun :-)

Name change
This one is easy, as long as you can remember your name of course!

0	1	2	3
Black	Navy	Green	Teal
4	5	6	7
Maroon	Purple	Olive	Silver
8	9	10	11
Grey	Blue	Lime	Aqua
12	13	14	15
Red	Pink	Yellow	White

Nice colours
Anything you like, anything, go on, pick a colour.

Way to go!

18

FUZE®

Project stage 1-2
"Variables eh?"

In this project you will learn;

To write a program including;
Conditional Loops
Displaying text
Variables
Delays

New commands introduced
CLS, END, REPEAT UNTIL Loop

Written by Jon Silvera

For educators:
Mapping with the Curriculum

Project 1-2 introduces simple numerical variables and conditional REPEAT LOOPS.

Using a loop to reduce the number of available apples each time one is eaten the program offers an insight into conditional UNTIL loops, more complex program flow and simple variable setting and manipulation.

Key Stages 1, 2 & 3

Project 1 - The basics

This stage introduces Variables. Variables are an essential part of every programming language, so pay attention!

Variables

Before we get started we should explain what variables are and why we use them in computer programming. The word itself means changeable or something that can or does change. The weather is a variable because it changes all the time.

First off make sure **FUZE BASIC** is ready to use. If it's not running already, switch on the **FUZE** and then double click the **FUZE BASIC** icon. When you see the **Ready>** prompt **[Pic 1]** we can begin.

Computer programs use variables to store information so we can access it whenever we need it. For example here are two variables;

Type in;

Girls=10 and press **Enter**
Boys=10 and press **Enter**

(*we **do not** want to be in the Editor for this part, just the start screen. This is also known as '**Direct Mode**' as we can type commands that will **RUN** straight away*)

Now we have stored information in the variables they will never change unless we do something to them. We no longer need to remember the numbers, just the names. We now know how many girls there are by looking at the variable "**Girls**". **[Pic 2]**

How long do you think I can keep these comments going ?

SCORE

Almost every game uses the variable name **SCORE** to store a player's score. When you do something good this happens; **SCORE=SCORE+POINTS.**
We can say that collecting a gold coin makes POINTS=100. It's not rocket science, well unless it's a game about rockets of course!

Case Sensitive

Case what? To a computer there's a big difference between **HAT** and **hat** and even **Hat**. Each one of these is seen by the computer as different and so they could be used as three different **variables**.
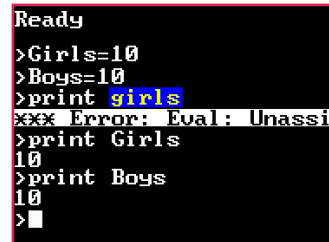
It is very important to remember this as **FUZE BASIC** will report an error if it does not recognise a **variable**. If we use **HAT**=10 and then say **Print hat**; we will get an error because the computer thinks it is a different variable that we have not yet given a value. Programming languages are generally **Sensitive** to the **Case** we use. **[Pic 2]** demonstrates this very well, can you see why we got an error?

20

Type in;

Print Girls and press **Enter**
Print Boys and press **Enter**

You should see 10 each time. Now we can do all kinds of clever things, type;

Class=Girls+Boys and press **Enter**
Print Class and press **Enter**

The answer should of course be 20. Now try;

Girls=Girls+5 and press **Enter**
Class=Girls+Boys and press **Enter**

What's going on here? We know **Girls** is equal to 10 so the first sum is 10+5, this means **Girls** is now equal to 15. Then we made **Class** equal **Girls+Boys** again, so now **Class** equals...? Well, ask the computer;

Type **Print Class** and press **Enter**; the answer will hopefully be 25

So far we've only stored numbers in our variables. These numbers can be whole numbers like 1, 2, 10, 200 or they can be numbers with a decimal point like 0.3, 2.67, 89.9999 or 0.001. This means we can use variables to work on very complex numbers. That's not all, we can even store words and sentences in variables, but we'll get to that later on. Let's start a new program!

Quite a while I think, if I really put my mind to it.

More about Variables

There are a few rules we need to be aware of when it comes to naming our variables. For example; A variable must not start with a number but can contain or end with one. They cannot include any spaces, hyphens or full stops. Basically a variable should not start with a number and should only contain the letters Aa to Zz and or the numbers 0 to 9. They can however include the underscore character "_"

Good	**Bad**
Blue_Cars=10	Blue Cars=10
RED_CARS=5	RED CARS=5
greencars=6	green.cars=6
boy1=14	1boy=14
Girl_2=12	2_Girl=12
Boys_and_Girls=20	Boys_&_Girls=20
Two4tea=1	2fortea=1
Pizza_Topping=2.50	Pizza-Topping=2.50

Some tips when using variables

Most variables are used many times in a single program. When you decide on a name for one it's therefore very sensible to make it as short as possible but at the same time easy to remember. For example; Player_One_Score=0 could be simply, P1_Scr=0. This takes a lot less typing but still makes sense.

Press **F2** to enter the Editor. If there is another program listing there then make sure it isn't needed and then press **F12** to clear it.

Start with the following few lines of code;

Editor
```
CLS
Apples=6
Print "We have "; Apples; " Apples"
END
```

Press **F3** to **RUN** the program **[Pic 3]**. The first time you do this you will be prompted to **SAVE**. Give it a name and press enter.

Firstly the program clears the screen, **CLS** and then stores the number 6 in the variable **Apples**.

The **Print** line now looks more complicated. We actually display three things, the words "We have " then the variable **Apples** and then the *word* " Apples". Remember, we use the semi colon to tell the computer to print the next item right after the last.

When the computer is asked to **Print** a variable it does not print the variable name, but its value. In this case, when the computer sees the variable **Apples,** it knows to print 6 as this is the value we gave to the variable in the line above.

Because the last word, " Apples" is in quotation marks, the computer does not see this as a variable, it is just text it must display.

And another pointless comment…

```
We have 6 Apples
* F2 -> Edit or ESC: ▮
```

[Pic 3]

Speaking of Apples...
The Apple Computer Company was founded on the 1st of April 1976 by three young men called Steve Jobs, Steve Wozniak "Woz" and Ronald Wayne. The first Apple computer was called the Apple I and was designed and built in Steve Jobs' garage. They sold just a few hundred of them.

The Apple II was shown at the West Coast Computer Faire in America on the 16th of April 1977. The orders started to flow and the company was, unknowingly, on the way to becoming one of the largest companies in the world today.

Not bad for three young men working from a garage. Actually the story goes, only the two Steves were really responsible as Ronald Wayne left just three months after it began, selling his stake in the company for just £500 - this would have been worth billions today!

So, what programming language do you think it came with?

BASIC, that's what, good old **BASIC**. While the language has come on a long way since the early versions, you'd be amazed just how similar **FUZE BASIC** is today.

Hmm.. All these apples are making me hungry! Edit your code to;

```
CLS
Apples=6
Print "We have "; Apples; " Apples"
Print
Print "If we eat one…"
Print
Apples = Apples - 1
Wait(2)
Print "then we have "; Apples; " left."
END
```

The extra **Print** command after each **Print** line displays a blank line to keeps things tidy.

Apples=Apples-1 instructs the computer to reduce the value of **Apples** by one and store the answer back in the variable **Apples**.

We have used a **Wait(2)** command to tell the computer to wait for two seconds. Well, we need time to eat it!

Finally we **Print** the result. This time the value of **Apples** will be different. **RUN** (**F3**) your program to see what happens **[Pic 4]**.

Still hungry huh? Well hold on to your seats, I can feel one of those **LOOP** things coming on. Return to the **Editor,** unless you're already there, by pressing **F2**, and edit your program to the following;

And another pointless comment…

GET WITH THE PROGRAM
Project Stage 1-2
"Variables eh?"

[Pic 4]

```
We have 6 Apples
If we eat one...
then we have 5 left.
× F2 -> Edit or ESC: █
```

Memory
Computers are all about memory. In fact without memory a computer would not really be a proper computer at all.

When a program is running it's stored in the computer's working memory (**RAM** - **R**andom **A**ccess **M**emory). If the power is switched off everything in **RAM** is lost. To avoid this we use storage memory to save our work. This could be on a hard disk, Solid-state drive, memory card or a USB drive These kinds of memory devices store information permanently without the need for power. Storage memory is much slower than **RAM** so we use both, otherwise computers would be very, very slow.

CPU - You have to love computing and all its acronyms. An acronym is a word made up of the first letters of a sentence or phrase. We use them to shorten a long phrase into a short one. For example "**PE**" is short for **P**hysical **E**ducation and "**PSHE**" for **P**ersonal, **S**ocial and **H**ealth **E**ducation.

It's a bit like **TXT** speak really... LOL!

So back to the **CPU** then. The **C**entral **P**rocessing **U**nit is quite simply, the brain of any computer. It's the boss - it keeps everything under control. A bit like a Head Teacher, but it doesn't moan all the time!

Editor

```
Apples=6
Cycle
CLS
Print "We have "; Apples; " Apples"
Print
Wait(1)
Print "If we eat one…"
Print
Apples = Apples - 1
Wait(2)
Print "then we have " ;Apples; "left."
Wait(2)
Repeat
END
```

Before we run the program let's take a quick look to see what's going to happen.

Remember the **Cycle** & **Repeat** commands from the last stage? Everything between these two lines will be repeated so when it gets to the **Repeat** command it jumps back to **Cycle**, clears the screen (**CLS**) and does it all again. Each time the value stored in the variable, **Apples,** is reduced by one.

RUN (**F3**) the program **[Pic 5]**. What happens when we get to zero Apples? We can fix that. **Exit** the program (**Esc**) and press **F2** to return to the editor.

And another pointless comment…

P.I.C.N.I.C.
Problem **I**n **C**hair, **N**ot **I**n **C**omputer. Don't forget, a computer will only do exactly what it is told to do, nothing more, nothing less. Because a variable is just a name we can store numbers in, it matters not if that number is large, small, positive (+) or negative (-).

[Pic 5]

If we tell a computer to count down in one's from 50 a hundred times, it will do exactly that. It won't stop at zero, it will just carry on until it gets to minus (-) 50 **[Pic 6].**

[Pic 6]

It's logical
Computers are all about logic. There are no grey areas. No maybe, perhaps or possibly. Either something is or it is not.

That's not to say we can't program a computer to behave illogically. In fact, it's possible to write programs that can appear human. There are even competitions to see who can write a program to fool a human being (that's one of us), into thinking a computer is actually another human. I recommend researching 'The Turing Test'. We'll come back to Turing, or Mr Alan Turing to you and me, a bit later as he was a very important chap indeed. I say 'was' because he's been dead for more than half a century, which is a shame!

Editor

```
Apples=6
Cycle
CLS
Print "We have "; Apples; " Apples"
Print
Wait(1)
Print "If we eat one…"
Print
Apples = Apples - 1
Wait(2)
Print "then we have ";Apples;" left."
Wait(2)
Repeat Until Apples = 0
CLS
Ink=Red
Print "We ran out of Apples!"
Print
Ink=White
END
```

Take a look at the **Repeat** command. We've added an **Until** condition to it. This says keep repeating the program **Until** the value of **Apples** is equal to zero. So every time the program arrives at this command it checks to see if **Apples=0** and if not it goes back and does it again. If it is zero then it goes on to the next line and carries on with the program.

When **Apples** gets to zero we clear the screen (**CLS**), change the text colour and **Print** a new message, and just to keep things tidy **Print** a blank line and change the text colour back to white ready to edit.

See!

LOOP the LOOP

Most computer programs do similar things over and over again so understanding **Loops** is going to be very important. Consider games for example. A game needs to check to see if you're pressing a key or moving a joystick. Even when you move a mouse around the screen or touch a screen to zoom a photo. The entire time, the computer is running in a loop checking the mouse for movement, the screen for touch and the keyboard for key presses.

Loops are another very important part of any computer program so once again… **PAY ATTENTION!**

In our Apple program we've used a specific kind of loop called **'Repeat Until'**

[Pic 7]

The beginning of the **Loop** starts with the **Cycle** command. Everything in between **'Cycle'** and **'Repeat Until'** is repeated until the **'Until'** condition is met. So the program will ony continue past this line when **Apples=0** otherwise it jumps back to the original **Cycle** command.

Before we end this stage we're going to look at how we enter our programs, because as they get bigger and more complicated they can start to look messy and become difficult to understand.

So far, we haven't worried about general formatting but from now on we will!

You don't need to enter the program below (you can if you like but remember to save your current work first). A quick glance and you can see how much easier it is to read.

Editor - **Example - you do not have to type this in**

```
CLS
CYCLE
  INK=Red
  PRINT "My name is Raspbery Pi"
  PRINT
  WAIT(1)
  INK=White
  PRINT "I live in this FUZE computer"
REPEAT
END
```

All commands are now in capitals so we can see instantly what is a command and what is a variable.

More important though is the use of **indentation** (nudged over!). Every time we use a **LOOP** the lines in the middle are indented. This makes everything much neater. Sorry to go on about it, but you'll find out later it's very important indeed - not like making your bed or getting to school on time, no, this stuff really matters!

Just a couple more and I'll have this one in the bag!

More about Memory, CPU's and Loops
The memory in a computer stores all the information a program needs to work. Imagine computer memory being like a ladder, a really, really tall ladder indeed. Each foot hole is like one unit of memory that can contain a single number.

If we store 10 numbers in memory and want to add 5 to each one, then firstly we need a **Loop** to count from 1 to 10. The **CPU** has to go to the first memory unit, copy the number to a temporary location, add 5 to it and then copy the result back into the first memory unit. The **loop** goes back to the beginning but this time works on the second memory location and so on, until it has completed the entire block of ten memory locations.

A photograph is stored in memory as a series of numbers. A high quality picture needs millions of memory units. To make the picture brighter we have to change every single number. This means the computer has to do millions of simple sums just to increase the brightness of a picture.

Luckily **CPU's** are very fast. In fact most modern computers have more than one CPU. You'll read about computers with Dual Core (2 CPU's), Quad Core (4), Oct Core (8) and higher - basically, the more Cores, the more powerful and the more things it can do in a '**clock cycle**'.

CPU speed is measured in Hertz (Hz), or at least it used to be. Today we measure in Mega Hertz (Mhz) and Giga Herts (Ghz). One Hz is one **clock cycle** per second. At one Hz the brightness calculation above would take weeks to finish. With Ghz CPU's the same process can be done millions of times

Stage 1-2 Assessments

So, you think you've grasped variables eh? Just you wait, you've seen nothing yet! For now though...

Assessment 1 ⭐

Can you change the program so that it's all about a different item of fruit. Don't forget to change all the variables and all the times we mention apples.

Assessment 2 ⭐

Can you change how many we eat at a time. Remember to change the text saying "if we eat one...", as well as the sum using the variable **Apples**, or whatever you changed it to above.

Assessment 3 ⭐

How about starting with a higher or lower number of items at the beginning.

Assessment 4 ⭐⭐⭐

Finally, try this; Start with 10 **Cakes**. Count down 2 at a time and when you are down to 4 left change it to say, "I'm bored of cakes... can I have an apple?!"

End of Stage 1-2

Hmm.. not that I've got a bag, doh!

```
We have 4 Pears
If we eat one...
then we have 3 left.
        We ran out of Pears!

        * F2 -> Edit or ESC: ▮
```

Apples are not the only fruit!
Remember to change 'Apples' in every instance.

```
We have 12 Pears
If we eat three...
then we have 9 left.
        We have 3 Pears
        If we eat three...
        then we have 0 left.
```

Make it pretty!
While you're at it why not liven things up with a bit of colour.

```
We have 6 Cakes
If we eat two...
then we have 4 left.

    I'm bored of cakes...
    can I have an apple!

 * F2 -> Edit or ESC: ▮
```

Now that's impressive
Ok, if you've managed to do this part you're doing really well.

Project stage 1-3
"It's String Theory - knot!"

In this project you will learn;

Nothing about string theory, but you
will discover something about;
Text based variables (strings $)
Conditional IF statements
How to accept input from the keyboard

New commands introduced
INPUT, IF THEN

Written by Jon Silvera

For educators:
Mapping with the Curriculum

Text based variables or strings ($) are essential
in programs to receive input from a user.

Once a $ variable is received it can be
examined and compared. Decisions (program
flow) can be influenced depending on input
received. "Yes" or "No" for example can
determine the path a program takes.

In this example the colour 'pink' is used to
choose a response from the computer.

Key Stages 2, 3 & 4

Project 1 - The basics

This stage introduces a new variable type called a **String** variable or a **$**. **String** (**$**) variables are used to store text information rather than numerical. The name **String**, comes from the variables containing a string of letters or characters.

So far we've used variables to store numbers, but what if we wanted to store information about people. For example, names and addresses or favourite films or bands. For this we need to store text information, not numbers.

Fortunately **FUZE BASIC**, and all other programming languages allow for this. With **BASIC** you simply add a '**$**' symbol to the variable name and this tells the computer this variable is for text.

Let's take a look. If it's not running already, switch on the **FUZE** and double click the **FUZE BASIC** icon. When you see the **Ready >** prompt we can begin.

As in the last stage we're going to type a few direct commands before we write a full program. Direct mode is great for experimenting. So... type in; **[Pic 1]**

Name1$="Bilbo" - press Enter
Name2$="Potter" - press Enter
Print Name1$ - press Enter
Print Name2$ - press Enter

In 2013, there were around 10 billion devices connected to the Internet

[Pic 1]

```
Ready
>Name1$="Bilbo"
>Name2$="Potter"
>
>Print Name1$
Bilbo
>
>Print Name2$
Potter
>
```

To "Quote.."
Whenever we store characters in a **$** (string) variable we must use quotation marks. This tells the computer it's text information and not just another variable name.

The contents of a **$** (string) variable can be pretty much anything although it's best not to use additional quotation marks within a $ text. **[Pic 2]** gives an example;

[Pic 2]

```
Ready
>a$="Peter said "Hi Bob""
xxx Syntax error trying to parse: ... Hi Bob""
>
>a$="Peter said 'Hi Bob'"
>
>print a$
Peter said 'Hi Bob'

>
```

As you can see using **"Hi Bob"** inside the **$** variable causes a Syntax error. Replacing them with apostrophes fixes the problem.

29

Type in;

```
Name3$=Name1$+Name2$
```
- press Enter
```
Print Name3$
```
- press Enter

This should give you the answer **BilboPotter**. That's a bit untidy, enter the following;

```
Name3$=Name1$+" "+Name2$
```
- press Enter
```
Print Name3$
```
- press Enter

Now you should see **Bilbo Potter** which looks much better but still, it's not quite right is it? Perhaps you should add another variable;

```
Name4$="Baggins"
```

This should sort things out, type in;

```
Name3$=Name1$+" "+Name4$
```
- press Enter
```
Print Name3$
```
- press Enter

Bilbo Baggins

There, that should keep everyone happy, especially Tolkien!

So text variables (**$**) can be treated in a similar way to number ones. We can add bits, take bits away, search for information in a **$** and even convert them into numbers.

Time to begin a new program!

This is incredible considering there are only 7.1 billion people in the world today

So what has the computer ever done for us?

Only a few years ago people used typewriters to write letters and the telephone to talk to other people, oh and there was no Internet! If you wanted to find out about something you had to look it up in a book. If you didn't have such a book and if there wasn't a good bookshop nearby then you would go to the library where they had thousands of books.

Using the telephone was a very different experience than it is today. Most households had just one phone so if you wanted a private conversation, off you'd go, rain or shine down the road to the nearest phone box (a big red cubicle in the middle of the street!). Sometimes you would have to queue and if the person you wanted to speak to wasn't in, you couldn't leave a message. You even had to insert coins before it would work!

Today we can text a friend in seconds, email letters, memos and pictures anywhere in the world and look up information on the Internet about anything you can imagine.

More than this though, it was not that long ago you would need a different device to do any of these things - now we just pick up our phone or tablet and do everything from there including take photos, listen to music and watch films.

It's very likely, with recent developments, that in the next few years all of the above is going to be on your wrist, straight from your watch and a pair of snazzy glasses!

Press **F2** to enter the Editor. If there's another program listed then make sure it isn't needed and then press **F12** to clear it.

Start with the following lines of code;

Editor
```
CLS
INPUT "Hi, what is your name? ", name$
WAIT (1)
CLS
PRINT "Hello "; name$
END
```

CLS clears the screen and sets the background with the current **PAPER** colour, which if it has not been set, is always Black.

The **WAIT (1)** command tells the computer to wait for one second. We can change the length of time by setting the number higher or lower. 1 is one second, 0.5 or .5 is half a second and 0.1 is one tenth of a second and so on. The **END** statement indicates the end of the program.

So, on with the program. We've introduced a new command, **INPUT**. When used like this the computer will display the text in quotes (**""**) and then wait for the User to 'Input' something. Whatever is typed in is then stored in the variable, **name$**

RUN (**F3**) and enter your name when asked then press **Enter [Pic 3]**

and only 30% of the world's population is actually connected to the Internet

GET WITH THE PROGRAM
Project Stage 1-3
"It's String Theory - knot!"

[Pic 3]

INPUT is used when information needs to be entered into a program. In fact, one way or another the **INPUT** command, or variations of it, are used every time you enter your name and email address on a web page, every time a PIN number is entered in a cash machine and even, every time a number is tapped into a phone. It is therefore worth spending some time getting to know it.

We can use **INPUT** to request numbers or text. The variable name at the end of the **INPUT** statement should have a **$** symbol to indicate it is a text variable. This is not required for a numerical one.

name$ = 10 - will give an error
name = "Taylor" - will give an error

31

It really doesn't matter what you type in as it will still be stored in the variable **name$**. Now let's add another variable and see what else we can do. Edit your code as follows;

```
CLS
INPUT "Hi, what is your name? ", name$
PRINT
INPUT "What is your favourite hobby? ", hobby$
PRINT
WAIT (1)
CLS
PRINT "Hello "; name$
PRINT
WAIT (1)
PRINT "So you like "; hobby$; " eh!"
END
```

We have a new question about hobbies and a new variable. The result is stored in **hobby$** and can now be used at any time just by referring to the variable name. Notice on the last **PRINT** line we've inserted the variable **hobby$** right into the middle of a sentence. Of course it won't display "**hobby$**" as it knows to display whatever is stored in the variable instead. This helps us to make the program seem more friendly and personal.

Let's find out a bit more about our user shall we? First though, now would be a good time to save our work - When in the Editor press **F5** to save your program. Then edit the program as shown overleaf.

This means on average, there are five devices connected to the Internet per person

[Pic 4]

There are many commands related to INPUT that will help you to enhance your programs. Most of these allow the computer to understand what has been entered so you can program it to respond accordingly using IF THEN commands.

One of the main computer uses over the last seventy years or so, has been to store information about people so it can then be found quickly by hospitals, banks, insurance companies and the police.

Editor

```
CLS
INPUT "Hi, what is your name? ", name$
PRINT
INPUT "What is your favourite hobby? ", hobby$
PRINT
WAIT (1)
CLS
PRINT "Hello "; name$
PRINT
WAIT (1)
PRINT "So you like "; hobby$; " eh!"
PRINT
WAIT (2)
INPUT "How old are you? ", Age
PRINT
PRINT "I see…"
PRINT
WAIT (2)
IF age < 10 THEN PRINT "Wow, so young, you're lucky!"
IF age >= 20 THEN PRINT "that's really, really old!"
PRINT
END
```

RUN (**F3**) the program. Clever eh! Run it again so you can enter ages younger than 10 and older than 20.

We've added a new variable, "**age**" to store the value the user enters and then the two "**IF THEN**" statements give different results depending on the user's input. Complete the program overleaf…

How many have you got?

"It's String Theory - knot!"

The Database

A database is quite simply a list, often a very large list, of information stored on a computer. A simple database can look like the following:

Pet Monster Database

Record 1:

[Field 1] Pet Name:	Tyrannosaurus Rex
[Field 2] Size (lrg-med-sml):	lrg
[Field 3] Cuteness (1 to 10):	3
[Field 4] Attack power (1 to 100):	95
[Field 5] Health (100 to 1000):	800
[Field 6] Speed (1 to 100):	20

Record 2:

[Field 1] Pet Name:	Fluffy the Rabbit
[Field 2] Size (lrg-med-sml):	sml
[Field 3] Cuteness (1 to 10):	10
[Field 4] Attack power (1 to 100):	10
[Field 5] Health (100 to 1000):	100
[Field 6] Speed (1 to 100):	95

Once the information has been entered we can write a program to perform different tasks depending on what we want to do. Say for example we wanted to make a card game, we could use the DATA (the name given to information stored in a database) to work out who was the strongest or fastest and therefore was most likely to win in a battle or win a race or even a beauty contest.

I think it's safe to say that the humble database is very likely to be the most common type of program ever.

```
CLS
INPUT "Hi, what is your name? ", name$
PRINT
INPUT "What is your favourite hobby? ", hobby$
PRINT
WAIT (2)
CLS
PRINT "Hello "; name$
PRINT
WAIT (1)
PRINT "So you like "; hobby$; " eh!"
PRINT
WAIT (2)
INPUT "How old are you? ", age
PRINT
PRINT "I see…"
PRINT
WAIT (2)
IF age < 10 THEN PRINT "Wow, so young, you're lucky!"
IF age >= 20 THEN PRINT "that's really, really old!"
PRINT
WAIT (2)
PRINT "Hey, what's your favourite colour "; name$;
INPUT col$
PRINT
WAIT (2)
IF col$<>"pink" THEN PRINT "shame, I'm not fond of ";col$
IF col$="pink" THEN PRINT "That's incredible, me too!"
PRINT
PRINT "Bye for now…"
END
```

I can count at least twenty three in my house, that can't be good, can it?

GET WITH THE PROGRAM
Project Stage 1-3
"It's String Theory - knot!"

Storage

In the early days of computing storage was a major headache. Developing new technology to permanently store a lot of data which can then be retrieved very quickly is big business.

The problem was, and still is today, that no matter how much storage we have, we still find ways of making the data larger and larger .

To understand this better we need to look at how information is stored on a computer. Take a deep breath, this is a lot to take in!

Take the letter 'A'. To a computer the letter A has no meaning. When it is stored on a hard drive or in the computer's memory it is actually stored as the number 65, B is therefore 66. A letter 'a' however is stored as number 97 and the number '1' as 49. Even the space symbol has a number, it's 32.

But, that's not the half of it as computers don't actually recognise these numbers either!

If you look closely, very, very closely at computer memory it is actually just a series of '0's and '1's. It's the arrangement of these digits that represent real numbers, letters and even pictures.

The use of '0's and '1's to represent more complex numbers was established in the late seventeenth century by Gottfried Leibniz and is referred to as the Binary Numeral System or Base 2.

It all looks a bit too simple, just zeros and ones in a long line but there's a lot of work required to turn this into information we can understand. See overleaf…

RUN (**F3**) the program. We now know quite a lot about our user but some of these new statements need explaining.

Firstly lets look at "**IF age < 10 THEN**" and "**IF age >= 20 THEN**";

These are called conditional **IF THEN** statements. If the condition is met then it will do whatever is after the **THEN** instruction. If the condition is not met then it will ignore the line and go on to the next one.

In this case the first condition is; "**IF age < 10**" or "**if age is less than 10**" then print the "so young" message. The next line checks the condition "**IF age >= 20**" or "**if age is more than or the same as 20**" then print the "old" message. "**>=**" and "**<=**" are known as "**more than or equal to**" and "**less than or equal to**"

IF THEN conditional statements are used a great deal in programming. We will be exploring them a lot more later.

The last part of our program asks the user, by name (with **name$** added to the end of the line) for their favourite colour and depending on the answer given, then displays different messages. The user's favourite colour is stored in the text variable **col$**.

We've also introduced another variation of the "**IF THEN**" conditional statement. This time we've used **IF col$ <> "pink"** and **IF col$ = "pink"**. "**<>**" means "**is not the same as or not equal to**" and "**=**" means "**is the same as or is equal to**".

Ok, that's all for now - please try the assessments overleaf.

Perhaps the more astonishing fact is not how many people *are* connected…

More about storage

A computer reads sequences of Binary digits and then software (programs) converts them into useable information.

Going back to our letter 'A', which as we now know is represented by the number 65 is stored in computer memory thus;

"**01000001**" So obvious you say, well yes, quite!

Ok, here's how it works; each digit from **right to left** has a value. The first is 1, the second is 2, and then 4, 8, 16, 32, 64 and 128.

Each individual binary digit is call a '**BIT**', eight in succession are called a '**BYTE**', 16 of them a '**WORD**', 32 a '**LONG WORD**'. We only need to worry about **BITS & BYTES** though (phew!)

So, '01000001' = 65 because;

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

We simply add the BITS that are on '**1**', together to give us our actual number, 65 in this case, and then the computer's Operating System works out the rest. So if it is expecting text it knows that 65 equals the letter 'A' and so it displays an 'A'.

Do you know what this Binary number represents?

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

We will come back to Storage and Memory in a later project.

Stage 1-3 Assessments

Right then, now that you've mastered String Theory, er.. not quite!, see how you get on with these;

Assessment 1 ⭐

Could you ask another question after the Hobby one? What about "Favourite food" using food$ as the variable name. You should also add something like **PRINT "You like to eat "; food$** too.

Assessment 2 ⭐

Using "**IF age = **", add one or two new responses to the "How old are you?" question. If you check, you will see that nothing currently happens if you enter numbers between 10 and 20...

Assessment 3 ⭐⭐⭐

Can you change the colour section so that it will respond with different answers to more colours. You should remove the line with **IF col$ <> "pink"** for the moment as this will confuse things.

End of Stage 1-3

```
Hi, what is your name? Jon
What is your favourite hobby? Counting clouds
What's your favourite food? Chocolate frogs█

Hello Jon
So you like Counting clouds eh!
and you like to eat Chocolate frogs
How old are you? █
```

Need INPUT
Turn your computer into a friend for life... so you'll always have someone to talk to!

```
INPUT "How old are you? ", age
PRINT
PRINT "I see..."
PRINT
WAIT (2)
IF age < 10 THEN PRINT "Wow, so young, aren't you lucky!"
IF age < 19 THEN PRINT "Yeah
IF age = 13 THEN PRINT "13 e
IF age >= 20 THEN PRINT "tha
PRINT
```

How old?
You know it's rude to ask old people their age so don't ask a teacher!

```
Hello Jon
So you like swimming with lug worms eh!
and you like to eat orange peel
How old are you? 19
I see...
Yeah, yeah, pull the other one!
Hey, what's your favourite colour Jon? █
```

```
Hello Gracie
So you like Minecraft eh!
and you like to eat Carrots
How old are you? 9
I see...
Wow, so young, aren't you lucky!
Hey, what's your favourite colour Gracie? crimson
Hmm.. very sophisticated!
Bye for now...
```

Did you get this far? You deserve a pat on the back!

Project stage 1-4
"It's all 1s and 0s to me!"

In this project you will learn;

Where the fun starts! and a bit about Binary and why it is important;
Binary numbers
Font manipulation
Designing and animating small graphics

New commands introduced
DEFCHAR, FONTSCALE, HVTAB, CHR$

Written by Jon Silvera

For educators:
Mapping with the Curriculum

The binary number system is the very lifeblood of all computing systems. In fact, if it doesn't count in binary then it's probably not a computer in the first place!

A Space Invader graphic is used to introduce a complex concept in a very accessible way.

Simple font manipulation and animations are also used to provide additional engagement and students are encouraged to design their own graphics. Graph paper, pens, pencils and erasers are prerequisite for this project.

Key Stages 2, 3, 4 and beyond

Everything you were afraid to know about computers but had to ask!

A computer understands two things, On and Off. That's it, that's all of it. So how does a computer send emails, play games, make music, show movies and surf the Internet?

Back on page 35 we explained how Binary digits, or BITS, are used to represent decimal numbers. Here's a quick refresher;

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

= 65

Each binary digit in sequence represents a value. The sequence above is an 8 **BIT** number or a **BYTE**. If all the BITs were on, or 1s, and you add them all together you have a decimal value of 255. 1+2+4+8+16+32+64+128 = 255. There's still one more value though and that's if all the BITs are off or zero. This is the same as the decimal value 0. This gives us 256 possible values for an 8 BIT number. So what does a 16 BIT number look like...

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

If you add all of these together then the largest number you can get is 65535 or 65536 variations including zero.

Early computers could only process information 8 BITs at a time. This meant they could only access 256 bytes of memory at a time or calculate small numbers or display very simple graphics. Actually, using clever techniques it is possible to join two 8 BIT bytes to make a 16 BIT one but it would take the computer longer to do this as it had to work on twice as many bytes at a time. Computers back then were quite slow so this just made them slower still.

So, the name given to the single, smallest computing unit is a BIT. For now at least…

Then came **16 BIT** computing and not long after **32 BIT** and now **64 BIT** is the standard.

Using the same numbering system you will find a 32 BIT number goes up to about 4 billion and 64 BITs, well just rest assured it's huge!

As time goes by computers become faster and faster. Not just because the number of BITs increases but the speed information moves around the computer increases and the number of calculations a computer can do in a single second also improves every year or so. We measure computer speed in **MIPS** or **M**illion **I**nstructions **P**er **S**econd.

As ever, back in the old days things started off very slowly. In fact in the early 1970's speeds hadn't reached a single **MIPS** (**M**illion **I**nstructions **P**er **S**econd). Today in 2014 we see affordable computers running 200,000 MIPS. That's 200 billion instructions per second.

Even the humble Apple iPhone is a 64 BIT computer capable of around 20,000 MIPS and that's just a phone!

So back to **BITS** & **BYTES**, 0s & 1s. How does a computer take a series of binary digits and turn them into something we can understand?

As we've covered earlier, a computer can only understand 1s and 0's. Actually it doesn't even know they are numbers. To a computer something is either on "1" or off "0". Electronic switches, billions of tiny electronic on off switches, make up the brain of a computer (the CPU) and all its memory. The switches are called Transistors. A Microsoft XBOX ONE has 5 billion for example.

Many scientists and electronic engineers believe the transistor is the most important development in the history of computing. Without it the iPhone, Samsung Galaxy, PS4, Nintendo 3DS and just about any other computing device you've ever heard of may well not exist today. Or at least if they did they'd not be as small or as fast!

Everything a computer does is an interpretation of the state of these transistors. If they are on they have a value of 1 and if they are off then 0.

Using the binary system these can be converted into decimal numbers which in turn can be turned into characters, graphics and sounds.

Take a Space Invader…

This how it might look on screen (left) but to the computer it is just a series of switches left in an on or off position (right)
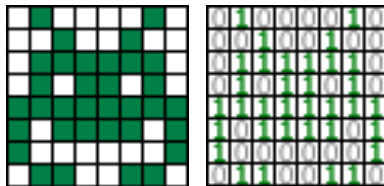
That's all well and good but it's difficult for us to use binary information like this. Using the binary system we can easily convert this into a series of decimal numbers. Remember, the first digit from right to left has a value of 1 then the next 2, then 4, 8, 16, 32, 64 and finally 128. At least that is what an 8 BIT number has. So going back to our chart we can convert this as follows;

Eight BITs in a row are known as a BYTE and 1024 BYTEs is a KILOBYTE while 1024 KILOBYTES is a MEGABYTE

128	64	32	16	8	4	2	1	
0	1	0	0	0	0	1	0	= 66
0	0	1	0	0	1	0	0	= 36
0	1	1	1	1	1	1	0	= 126
0	1	0	1	1	0	1	0	= 90
1	1	1	1	1	1	1	1	= 255
1	0	1	1	1	1	0	1	= 189
1	0	0	0	0	0	0	1	= 129
0	1	1	0	0	1	1	0	= 102

Now all we have to do is feed this information into the computer somehow. Sounds like we'll need some kind of Program!

All the letters, numbers and symbols we use when typing an email or a txt etc., are created in a similar way and we can in fact design our own characters using FUZE BASIC.

You will however need some graph paper, a pencil and probably a calculator but more on this in a minute…

In the **FUZE BASIC** editor type in the following program;

```
CLS
DEFCHAR (1, 0, 66, 36, 126, 90, 255, 189,129, 102, 0)
CYCLE
HVTAB (0,0)
PRINT chr$(1);
REPEAT
END
```

RUN, **F3**, this and you should see a tiny little invader at the top left of your screen. Press the **Esc** key to stop it. This won't do, it needs to be much bigger for a start and how about we bring our alien to life with some animation...

```
CLS
DEFCHAR (1, 0, 66, 36, 126, 90, 255, 189,129, 102, 0)
DEFCHAR (2, 0, 24, 36, 126, 90, 255, 255,66, 60, 0)
FONTSCALE (8,8)
INK = GREEN
CYCLE
HVTAB (0,0)
PRINT chr$(1);
WAIT(0.2)
HVTAB (0,0)
PRINT chr$(2);
WAIT(0.2)
REPEAT
END
```

1024 MEGABYTES are called a GIGABYTE, and 1024 of those, a TERABYTE and then

GET WITH THE PROGRAM
Project stage 1-4
"It's all 1s and 0s to me!"

This needs some explaining...

Firstly there is a new command to take care of, **DEFCHAR**. This allows us to redefine the shape of any of the standard characters. The first number, in this case 1 and 2 tells the computer we want to redefine the first two characters available.

The next ten numbers represent each horizontal line of the character. We can have ten lines down but I have kept the first and last as 0 so there's a gap on the top and bottom.

FONTSCALE determines how big the width and height of our text characters will be. In this case we have gone with eight times the normal size. Experiment using much higher numbers like 20 and 30. If nothing appears, then you've gone too big.

HVTAB allows us to position the text cursor (position) anywhere on the screen. We have set this at 0,0 or the top left of the screen.

We then have a simple loop with a couple of wait commands so it's not too fast (try removing or changing the values).
Using this method it is possible to create all kinds of simple graphic games and characters. There are limits though as we can only use a single colour and we can't really make anything detailed. That's Ok though as there other things we will cover later that will allow us to be much more creative.

In the **FUZE BASIC** editor change your program or start again to make it as follows; Note the numbers have changed a lot!

```
CLS
DEFCHAR (1, 126, 126, 24, 24, 24, 24, 24,24, 126, 126)
DEFCHAR (2, 0, 36, 126, 255, 255, 255, 126,60, 24, 0)
DEFCHAR (3, 0, 66, 36, 126, 90, 255, 189,129, 102, 0)
FONTSCALE (8,8)
CYCLE
INK = YELLOW
HVTAB (0,0)
PRINT chr$(1);
WAIT(0.5)
HVTAB (0,0)
INK = PINK
PRINT chr$(2);
WAIT(0.5)
HVTAB (0,0)
INK = GREEN
PRINT chr$(3);
WAIT(0.5)
REPEAT
END
```

RUN, **F3**, and all going well you should see an animation playing I - Love - Space Invader. We now have three steps to our animation but of course we could have any number.

You can now change any of these characters by simply changing the numbers.

a PETABYTE, an EXABYTE, a ZETTABYTE and finanlly a YOTTABYTE follow…

GET WITH THE PROGRAM
Project stage 1-4
"It's all 1s and 0s to me!"

Please grab some graph paper, a pen and a pencil and, if you are terrible at maths then maybe a calculator too. Draw the following grid pattern, **with a pen**, with the numbers along the top. You want 8 columns across and 10 rows down.

128	64	32	16	8	4	2	1

Then using a pencil, lightly fill in boxes to build your own characters, shapes or symbols. Add up the boxes in each row to get your decimal value at the end. You can then add these to your program by replacing the ones already there or create new ones.

DEFCHAR (#, specifies the character number that you use in the PRINT chr$(#) command. Be creative, although it is not easy with a small number of pixels to work with…

Stage 1-4 Assessments

Ok, you've learnt some important stuff here. Everything you know is 0 and 1s, but lets see if you've really got it....

Assessment 1 ⭐

Change the colours and the delays we used to make the animation show faster or slower.

Assessment 2 ⭐

Change one, or all, of the characters for ones you have designed yourself on graph paper. You will need to replace the numbers in the **DEFCHAR** command but remember, the first number tells the computer which character to replace. If you want to add more then you will need to increase this number.

```
DEFCHAR (1, xxx, xxx, xxx, xxx, xxx, xxx, xxx, xxx, xxx, xxx)
```

Assessment 3 ⭐⭐⭐

This one is a bit harder. Can you change the position on the screen of the character. Experiment to see how changing the values in **HVTAB (0,0)** make a difference to the location. Remember the first number changes the horizontal position and the second number the vertical position. The characters don't even need to be in the same position each time.....

End of Stage 1-4

But did you know that the name given to just 4 BITs is a NIBBLE? Comic genius' those geeks!
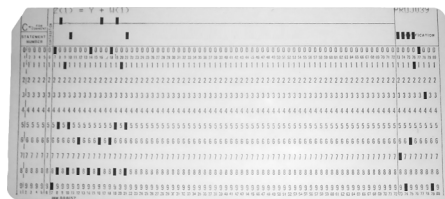

GET WITH THE PROGRAM
Project stage 1-4
"It's all 1s and 0s to me!"

More binary information

Many of the earliest computers used cardboard to store information. Strips and strips of card with little holes punched through them were used to feed information into a computer. They became known as 'Punch Cards'

Not all punch cards used the same methods but binary was certainly one of them. The cards were fed into a machine reader that scans for holes in the card. If there is a hole then it returns a value of 1 if there is no hole then a value of 0 is returned.

This series of 1s and 0s are then converted into machine code. Machine code is a very complicated programming language but is, in principle, still very similar to all other languages, including FUZE BASIC.

Today we use very similar methods to store information. The Compact Disc (CD), DVD and Blu Ray discs all use the same method. Tiny bumps are made on the inner surfaces of the disc by firing a laser beam. Again, a laser is used to read information back. A 1 or 0 is returned depending on there being a bump or not.

It's hard to imagine that nothing more than a long, long line of zeros and ones could possibly end up as the latest Taylor Swift song, but there you go, anything's possible.

FUZE®

Project stage
"Round Up"

In this project you will review;

Everything we have covered so far but with a much larger program;

Commands used
CYCLE, INK, PAPER, CLS, INPUT, WAIT, IF THEN ELSE, ENDIF, FONTSCALE, HVTAB, CHR$, RND, FOR, REPEAT, UNTIL, END, OR, TWIDTH, THEIGHT, LEN

Written by Jon Silvera

For educators:
Mapping with the Curriculum

While everything up till now is included, this project raises the complexity level by expanding simple loops into conditional ones, IF into IF THEN ELSE and ENDIF statements and, random number generation with RND.

The entire program serves a real purpose and presentation plays an important role too. Text styling and layout is introduced and the program flows with a beginning, middle and end to arrive at its purpose.

Fasten seatbelts, this is the real thing!

Key Stages 2, 3, 4 and beyond

Time for review

The last few projects have introduced you to a few programming fundamentals. However, we've not put them all together yet.

This project brings everything we have looked at so far into one fun little game. 'Think of a number' is a computing classic. I warn you though, the Round Up projects are generally a bit longer, so please be sure to save your work regularly and try very hard not to make any typing errors as they are harder to find in larger programs.

Also our methods change a little for the Round Up Projects, as again the programs are much longer. We will start, in this case, with a complete game, but it has no frills - those we will add one by one.

To the Editor we go

You need to start a fresh new program so first off clear anything you might already have in memory. If you're in the **Editor**, press **F2** to go to Direct mode. If you're already in **Direct** mode then save anything you need to and then type New and press Enter. Now press **F2** to go back to the blank editor and enter the following program;

Editor	press F2 to display the editor

```
CLS
PRINT "I am thinking of a number between 1 and 100"
PRINT "You have TEN turns to try and guess my number"
WAIT (1)
result$ = ""
number = RND (100) + 1
```

"Gangnam Style" by Psy is still the most viewed YouTube video of all time. It's been viewed over two billion times

```
turn = 10
guess = 0
CYCLE
  CLS
  IF turn < 10 THEN
    HVTAB (1, 1)
    PRINT result$
  ENDIF
  HVTAB (1, 3)
  PRINT "You have ";  turn;  " turns left"
  HVTAB (1, 5)
  INPUT "What is your guess ? ", guess
  WAIT (1)
  IF guess < number THEN result$ = "too low"
  IF guess > number THEN result$ = "too high"
  WAIT (1)
  turn = turn - 1
  IF turn < 1 THEN
    PRINT "Bad luck!"
    END
  ENDIF
REPEAT UNTIL guess = number
CLS
PRINT "WELL DONE"
END
```

First things first, save your work. In the **Editor** press **F5** to save. Then we can **RUN F3** and see how it plays.

Play the game through a few times and make sure you win and lose so you can see what happens.

Before we jazz things up a bit lets go through and see if we can make sense of it.

The first few lines are easy, **CLS, PRINT** and a **WAIT** command. These clear the screen, print a couple of lines of text and then pause for one second.

In the next few lines we define a few important variables that we are going to use throughout our game. **'result$'** is a string (text) variable to store 'too high' or 'too low' in depending on your guess.

'number' is the computer's guess. **RND (100)+1** picks a random number from zero to 100. Actually it is from 0 to 99.9999999 so it will never pick 100. We add 1 to the result so the **'number'** is then from 1 to 100.

'turn' is the number of player turns left and **'guess'** is the variable we store your guess in each time.

The **CYCLE** command starts our main loop. This loop is very important as it determines the flow of our program. Look for the line '**REPEAT UNTIL guess = number**'. We know that the variables '**number**' and '**guess**' are where we have stored the computer's choice and our guess.

The command '**REPEAT UNTIL guess = number**' will always return to the **CYCLE** command unless (**UNTIL**) our '**guess**' is the same as (**or equal to**) the computer's '**number**'.

Once we guess the right number then this condition is met and the program is allowed to pass the **REPEAT** and go on to the next part where it clears the screen (**CLS**) and prints a message.

Just before this, the **turn** variable is reduced by one and if it gets to zero (or less than one) then the message "Bad Luck!" is displayed and the program **ENDs**

If the correct guess is entered **guess = number** then the loop condition is met and the loop will exit and the message "WELL DONE" is displayed instead.

To make sure you understand how the program is working try changing the number of turns allowed to a smaller one and change the messages displayed as well.

What's in a name…? Amazon was initially named 'Cadabra', Twitch and Jitter were considered for Twitter and eBay actually stands for Echobay which could have been its name.

45

So, what do you think so far?

Be truthful, it's a bit dull isn't it?... Well, with a bit of tweaking we can change that. A bit of colour and tidying up will work wonders.

The problem is that we are about to make so many changes it is better if we start again. If you are in the editor then press **F2** to enter Direct mode and then type the following or if you are already in Direct mode just type it straight away

NEW (and then press enter)

Press **F2** to return to the editor and type in the following program;

Editor   press F2 to display the editor

```
FONTSCALE (3, 3)
  PAPER = Navy
  INK = Lime
  CLS
  HVTAB (1, 1)
  INPUT "What is your name? ", name$
  WAIT (1)
  PAPER = Maroon
  INK = Pink
  CLS
  HVTAB (1, 1)
  PRINT name$;
  INPUT ", would you like to play a game (y/n)? ", A$
  IF A$ <> "y" AND A$ <> "Y" THEN
```

```
    WAIT (1)
    PAPER = Black
    INK = Red
    CLS
    HVTAB (1, 1)
    PRINT "I'm very sorry to hear that... Goodbye"
    PRINT
    WAIT (3)
    CLS
    END
ENDIF
```

Ok, so this is just the start of our new version. When you **RUN** it you will see it is more colourful and personal, human even. It now asks for your name and if you would like to play a game.

If you don't want to play it says Goodbye. When you input the **Y / N** answer it stores your entry into the string variable **A$** and then checks to see your if answer was anything else other than an upper or lower case **Y**. If it's not a **Y** then it prints the Goodbye message but if it is then it reaches the end of the program.

It took just four years for the Internet to achieve 50 million users. Radio took 38 years to achieve the same and television 13 years. Today more than 2 billion people use the Internet!

Now we need to add the next part. After the last line, **ENDIF** please enter the following statements;

Editor   press F2 to display the editor

```
WAIT (1)
FONTSCALE (2, 2)
PAPER = Olive
INK = Black
CLS
HVTAB (1, 1)
PRINT "Great, then let's get started.";
PRINT
HVTAB (1, 4)
PRINT "I am thinking of a number between 1 and 100.";
WAIT (2)
HVTAB (1, 6)
PRINT "You have TEN goes to try and guess my number."
WAIT (3)
FONTSCALE (4, 4)
HVTAB (1, 5)
INK = White
PRINT "READY ..."
WAIT (3)
FONTSCALE (2, 2)
result$ = ""
number = RND (100) + 1
goes = 10
guess = 0
```

➡ A change of font and colours and we're on to the next part of our game.

Nothing clever happens yet, just a few more questions.

At the end we setup our main variables with;

result=""
number = RND (100) +1
goes = 10
guess = 0

The next part is huge so it would be a good idea to save and test what you have so far, just so you know it's all working as it should be.

As the program gets longer and more complex it becomes much harder to track down any errors or bugs that might have crept in.

When you're good to go please enter the listing on the next page immediately after the last line (guess=10)

There are now more than 1.1 billion registered websites

Editor	press F2 to display the editor

```
CYCLE
    FONTSCALE (3, 3)
    PAPER = Blue
    CLS
    IF goes < 10 THEN
      HVTAB (1, 1)
      INK = Pink
      PRINT "Your last guess with ";  guess;  " was ";  result$
    ENDIF
    INK = Silver
    HVTAB (1, 3)
    PRINT "You have ";  goes;  " goes left"
    HVTAB (1, 5)
    INPUT "Your guess this time ? ", guess
    IF guess < 1 OR guess > 100 THEN
      HVTAB (1, 7)
      PRINT guess;  "! I said between 1 and 100 silly"
      result$ = "silly!"
      WAIT (2)
    ELSE
      IF guess < number THEN result$ = "too low"
      IF guess > number THEN result$ = "too high"
    ENDIF
  ENDIF
  WAIT (1)
  goes = goes - 1
  IF goes < 1 THEN
    FONTSCALE (10, 10)
    PAPER = Black
    INK = Red
    FOR num = 1 TO 100 CYCLE
      PRINT "LOSER ";
    REPEAT
    END
  ENDIF
REPEAT UNTIL number = guess
```

The inventor of the modern World Wide Web, Sir Tim Berners-Lee, was knighted in 2004 by Queen Elizabeth II

→ The **CYCLE** command indicates the start of our Main Loop. This time however we are checking to see if the **guess** you entered is within the range of 1 to 100 and if it is either lower than 1 or higher than 100 then the message "**! I said between 1 and 100 silly**" is displayed.

The variable **result$** is used to store your clue for the next turn (**IF guess** is lower than the computer's number then **result$**="too low" - **IF guess** is higher then **result$**="too high" or if your **guess** is out of bounds then **result$**="silly!")

Each turn the variable **turn** is reduced by one until it finally will be below 1 where the "**LOSER**" message is displayed and the program **ENDs**.

This is the longest program we have written so far so once again please SAVE it and test it a few times. You should try and lose as well as win so you can make sure it does the right thing. Also try numbers below 1 and above 100 to make sure this part works too.

Now please pay attention as I shall say this only once...

In this next part, you do not need to add the grey parts. Just make sure you add the **CYCLE** command at the very beginning and the rest after the **REPEAT UNTIL number = guess** line.

```
Editor                    press F2 to display the editor
CYCLE      -add this before the first line
  fontScale (3, 3)
  PAPER = Navy
  …...
  …...      -You don't need to add the grey lines
  …...
    END
  ENDIF
REPEAT UNTIL number = guess
FONTSCALE (5, 5)    -add the rest after the last line
PAPER = Black
CLS
text$ = "WELL DONE"
FOR num = 1 TO 100 CYCLE
  INK = RND (15)
  HVTAB (TWIDTH / 2 - LEN (text$) / 2, THEIGHT / 2 - 2)
  PRINT text$;
  HVTAB (TWIDTH / 2 - LEN (name$) / 2, THEIGHT / 2)
  PRINT name$;
  HVTAB (RND (TWIDTH - 1), RND (THEIGHT))
  PRINT 10 - goes;
REPEAT
REPEAT
END
```

Now we have a proper ending. The **WELL DONE** message is only ever displayed if your **guess** is equal to the computer's **number**. However, the end sequence is tricky and needs some explaining; Firstly the words "**WELL DONE**" are stored in a variable called text$

There are more than 2,000,000 emails sent every second and more than two thirds of it is spam and junk mail.

Then a **FOR** loop is started. The loop will count from 1 to 100 so the loop runs 100 times.

A random (**RND**) colour from 15 is chosen.

Now **HVTAB** will position the text cursor at position X (horizontal), Y (vertical).

We want to print the message in the exact centre of the screen so we use a couple of system variables, these are ones that are looked after by **FUZE BASIC** itself.

TWIDTH and **THEIGHT** refer to the number of character positions there are across and top to bottom. We divide these by 2 to work out the middle. Then the **LEN** command tells us the length of **text$** and we divide this by 2 also.

We subtract half the length of t**ext$** from the middle point to give us the starting point of our **text$** message and **PRINT** it there. We then do the same for **name$** and print that in the middle also.

THEIGHT is also divided by 2 to get the vertical screen centre. Then a **RND** (random) position is chosen from the total screen width and height and the number of goes remaining (**10 - goes** tells us how many turns are left out of 10).

That's it. **RUN** it, **PLAY** it, **CHANGE** it and **REPEAT**!

GET WITH THE PROGRAM

FUZE®

Project stage 2-1
"True OR False?"

In this project you will learn;

About Boolean logic;
What is TRUE?
What is FALSE?
Variations in programming languages
AND, OR & NOT…. oh good grief?
Complex conditional IF THEN ELSE
statement.

New commands introduced
TRUE, FALSE, AND

Written by Colin Bodley

For educators:
Mapping with the Curriculum

This is a fairly difficult stage as Boolean Logic
needs a fair amount of explanation and can be
a hard concept to grasp.

If you are Key Stage 3 or above then it is
recommended that additional resources are
used to gain a thorough understanding of the
topic as Boolean operators like AND, OR, NOT
are essential in advanced programming.

Key Stages 2, 3, 4 and beyond

What is Truth?

This is a very deep question that humankind has been thinking about for thousands of years but I hope that you would agree with me if I told you that **1 + 1 = 2 ***

This is what is known as a **TRUE** statement. What the **=** (or equals sign) does is says that the things on the left hand side have the same value as the things on the right.

I hope that you would also agree with me if I told you that **2 + 2 = 5** is a **FALSE** statement, because the things on the left hand side do not have the same value as those on the right.

However the statement **2 + 2 <> 5** is actually a **TRUE** statement because **<>** means "Is not equal to" that is the things on the left hand side do **NOT** have the same value as the things on the right.

You may have noticed by now that **TRUE** and **FALSE** are exact opposites of each other, that is **TRUE** = **NOT FALSE**.

So, how do we use all of this in a computer program? Well, say that we have 2 numeric (number) variables called **X** and **Y** and I tell you that **X + Y = 2**. Is this a **TRUE** statement or a **FALSE** one?

It will depend upon the values of **X** and **Y.** If they both have the value 1 then it will be **TRUE**. If they both have the value 2 then it will be **FALSE** (2 + 2 = 4 not 2).

* This is actually more difficult to prove than you might think!

Boolean Variables

These are special types of variables in some programming languages that can only have one of two values: **TRUE** or **FALSE**.

In **FUZE BASIC** a normal numeric variable is used to do this with the value 0 (zero) representing **FALSE** and any number other than 0 (for example 1) representing **TRUE**. It also has two built in constants called **FALSE** and **TRUE** which have the values 0 and 1. You cannot change the values of these so for example **TRUE = 0** will give an error.

Conditions

A condition is a test of whether something is **TRUE** or **FALSE**. For example X = 2 will be **TRUE** if X has the value 2 but **FALSE** if it has any other value.

Conditional Statements

These allow a program to carry out different actions depending on the results of the condition. Without this our programs would always do the same things and would not be very interesting!

The IF…THEN Command

The simplest conditional statement is the **IF THEN** command which looks something like this:

IF *condition* **THEN**
  commands
ENDIF

If the *condition* is **TRUE** then the *commands* (there can be more than one) will be **RUN** but if it is **FALSE** they won't be. The **ENDIF** command marks the end of the commands to be **RUN** (or not!).

Right let's get programming. I assume by now that you know how to get **FUZE BASIC** up and running but if you can't remember you need to look back at the first project card (**1-1**).

Once you have done so, press **F2** to start the Editor. If there's another program listed then make sure it isn't needed and then press **F12** to clear it. Then enter the following lines of code:

Editor
```
X=1
Y=1
IF X + Y = 2 THEN
   PRINT X;" + ";Y;" = 2"
ENDIF
END
```

What do you think will happen when the program runs? Press **F3** to **RUN** it now and find out. As before you will be prompted to first **SAVE** your work. Enter a name for your program, for example IF THEN.

Be careful with this logic stuff! You might prove that black is white and get run over on a zebra crossing!

Programming Languages

There are lots of different programming languages and they all have different commands and ways of doing things. However, nearly all will have something similar to the **IF THEN** conditional statement on the left. Here are a few examples:

Python
if *condition*:
    commands

Pascal/Delphi
if (*condition*) then
begin
    commands
end;

C/C++/C#/Java/PHP
if (*condition*)
{
    commands
}

BASIC/Visual Basic
IF *condition* THEN
    commands
ENDIF

Don't worry too much about the details of each language. The most important thing is that you understand what a conditional statement is: one that is run when a certain condition is met.

Did you guess correctly? You should have seen the statement **1 + 1 = 2** printed on the screen because the condition **X + Y = 2** is **TRUE** when **X** is 1 and **Y** is 1 (1 + 1 = 2 is **TRUE**).

Now change the value of **X** and/or **Y** to something bigger than 1 and see what happens when you run the program again. This time you should get nothing printed to the screen at all. This is because the condition is now **FALSE**.

What if we want to print something out in this case? We could use two **IF** commands one testing if the condition is **TRUE** and the other if it is **FALSE**. Luckily we don't have to. To make our life easier there is another part to the **IF** command called the **ELSE** part.

The IF…THEN…ELSE Command

This is just an extension of the IF…THEN command and looks like this:

IF *condition* **THEN**
  commands
ELSE
  commands
ENDIF

In this case **IF** the c*ondition* is **TRUE** then the c*ommands* under **THEN** will be run until the **ELSE** is reached and if it is **FALSE** the c*ommands* under **ELSE** will be run until the **ENDIF** is reached. Now we can change our program to do different things depending on the values of **X** and **Y** as shown overleaf:

If you think this stuff is funny you need to check out the Hitch-Hikers Guide to the Galaxy!

Flowcharts

These are diagrams that show the way that a process flows. In the example below for the **IF…THEN…ELSE** command the program will either go down the right side or the left side depending on the outcome of the test of the *Condition*.
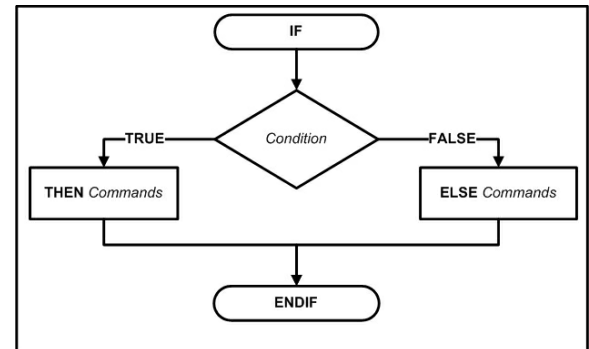
The different shaped boxes have different meanings. The rounded ones mark the start and end of the process, the diamond shape is a decision to go one way or another and the boxes are actions to carry out.

```
X = 2
Y = 2
IF X + Y = 2 THEN
   PRINT X;" + ";Y;" = 2"
ELSE
   PRINT X;" + ";Y;" <> 2"
ENDIF
END
```

This time when you run the program you should see the statement **2 + 2 <> 2** (remember that **<>** means is not equal to) as 2 + 2 is not equal to 2 and the **ELSE** part of the conditional statement is **RUN**.

So we now have a program that tells us that 2 + 2 is not equal to 2. This doesn't seem very useful! Let's make it a bit more general. Edit the program as shown below:

Editor

```
INPUT "X = ", X
INPUT "Y = ", Y
INPUT "Z = ", Z
IF X + Y = Z THEN
   PRINT X;" + ";Y;" = ";Z
ELSE
   PRINT X;" + ";Y;" <> ";Z
ENDIF
END
```

"We don't need no education" is a double negative which is a positive!

GET WITH THE PROGRAM
Project stage 2-1
"True OR False?"

NOT!
As mentioned earlier **TRUE** and **FALSE** are exact opposites of each other and the **NOT** operator will reverse them so:
TRUE = NOT FALSE and **FALSE = NOT TRUE**

So, for example, this means that the condition **A <> B** is the same as **NOT (A = B)**.

In some programming languages the **NOT** operator is written as **!** and the **<>** operator is written as **!=**

NOT Gate
In digital electronics a **NOT** gate takes an input signal which is at high (on, 1, +5V or TRUE) and outputs a signal that is at low (off, 0, 0V or FALSE) and the other way around.

Input	Output
1	0
0	1

You might use a **NOT** gate to switch a light on in a car when the door (and a switch attached to it) is open. When the door (and switch) is closed the input goes to on and the output (and light) goes off.

This time when you run the program it will prompt you to enter 3 numbers (**X**, **Y** and **Z** - press the **Enter** key between each) and will then tell you if **X** added to **Y** gives the result **Z** (or **NOT**). Sample outputs from the program are shown in **[Pic 1]** and **[Pic 2]** on the right.

String Comparison

Conditional statements work with string variables too. Two strings are considered equal if they have the same length and all of the characters match **EXACTLY**. This means that letters have to be the same case (capital letter or small ones) and punctuation has to be the same as well (including spaces).

Say that we want to write a program that prints a different message depending on who is running it, we could do something like this:

Editor
```
INPUT "Enter your Name: ", Name$
IF Name$ = "Dave" THEN
  PRINT "Hi Dave. Welcome back."
ELSE
  PRINT "I don't know you, "; Name$
ENDIF
END
```
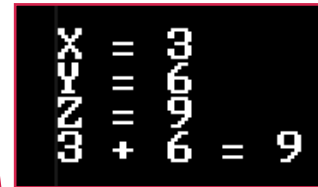
If you run this program and enter the name **Dave** (followed by the Enter key) you will get the output shown in **[Pic 3]**. If you enter any other name, including **DAVE** (all in capitals) you will get something like the output in **[Pic 4]**. You need to be very careful about case (whether letters are capital or small) when comparing strings.

Keep going!
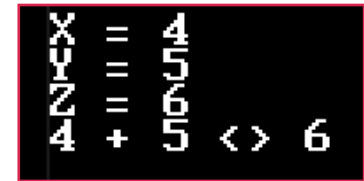
[Pic 1]

[Pic 2]

[Pic 3]

[Pic 4]

Multiple Conditions

What if you wanted to test more than one condition at a time? You could do something like this:

IF *condition1* **THEN**
  IF *condition2* **THEN**
   commands
  ENDIF
ENDIF

The *commands* would only be executed if *condition1* is **TRUE AND** *condition2* is **TRUE**. The **AND** operator makes this much clearer to read and understand:

IF *condition1* **AND** *condition2* **THEN**
  commands
ENDIF

The condition (*condition1* **AND** *condition2)* results in either **TRUE** or **FALSE** depending on whether *condition1* and *condition2* are **TRUE** or **FALSE**. If both of them are **TRUE** the result is **TRUE** but if either (or both) of them are **FALSE** the result is **FALSE**.

You can also test more then 2 conditions at the same time by using multiple **AND** operators e.g. (*condition1* **AND** *condition2* **AND** *condition3).* An example of using the **AND** operator is shown on the next page.

Nothing to see here, move along now!

### AND Gate

In digital electronics an **AND** gate takes two input signals which are either at high (on, 1, +5V or TRUE) or low (off, 0, 0V or FALSE). If they are both high then the output is high otherwise it is low.

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Symbol

An example of where you might use an **AND** gate would be in a microwave oven. The door could be made to open and close a switch attached to input **A**. Then the actual On/Off switch could be attached to input **B** and the oven would only work when the door was closed.

In this example the **<** operator means 'less than', **<=** means 'less than or equal to', **>** means 'greater than' and **>=** means 'greater than or equal to'. Enter the program into the **Editor** and **RUN** it. Try inputting different ages. Can you work out what answer you will get before you run the program?

Editor

```
INPUT "Enter Your Age: ", Age
IF Age <= 12 THEN
   PRINT "You are really young"
ENDIF
IF Age > 12 AND Age < 20 THEN
   PRINT "You are a teenager"
ENDIF
IF Age >= 20 THEN
   PRINT "You are really old"
ENDIF
END
```

What if we want to test whether one condition is **TRUE OR** another is **TRUE**. In this case we can use the **OR** operator as shown below:

IF *condition1* **OR** *condition2* **THEN**

   commands

ENDIF

In this case as long as one of the conditions is **TRUE** the commands will be **RUN** even if the other is **FALSE**. Only if they are both **FALSE** will they not be **RUN**.

The wheels on the bus go round and round, round and round, round and round. Come on now all sing together…. The wheels on the….

OR Gate

In digital electronics an **OR** gate takes two input signals which are either at high (on, 1, +5V or TRUE) or low (off, 0, 0V or FALSE). If either are high then the output is high otherwise it is low.

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Symbol
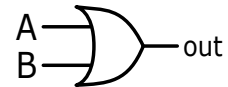
An example of where you might use an **OR** gate would be in a burglar alarm system. Sensors could be attached to doors and windows and attached to the **OR** gates inputs. If any one of the sensors is triggered then the output will go to on setting off an alarm.

Stage 2-1 Assessments
OK so you think you know what is **TRUE** and what is **FALSE**. Let us find out now.

Assessment 1 ⭐
Change the program on Page 4 to tell you whether two numbers when multiplied together equals a third number. The multiplication operator is ***** (asterisk - shift key and 8 on the keyboard) as x would get confused with a variable called x.

Assessment 2 ⭐
Change the program on Page 5 to give you a friendlier greeting than anyone else who is running the program.

Assessment 3 ⭐
Can you get this program to work if you enter your name in all capitals **OR** all small letters as well?

Assessment 4 ⭐⭐⭐
Change the program on the previous page to give different responses if you are between 20 and 30 or 30 and 40 etc.

GET WITH THE PROGRAM
Project stage 2-1
"True OR False?"

Summary
After completing this Project Card you should now understand what a Conditional Statement is: one that is only executed (**RUN**) under certain conditions.

A condition is a test of whether something is **TRUE** or **FALSE.** You can do this test using a number of operators:

Symbol	Meaning	= TRUE	=FALSE
=	Equal	2+ 2= 4	6 *9 = 42
<>	Not Equal	6 * 9 <> 42	2 + 2 <> 4
>	Greater Than	38 > 27	27 > 38
<	Less Than	27 < 38	38 < 27
>=	Greater Than or Equal	32 >= 32	31 >= 32
<=	Less Than or Equal	32 <= 32	33 <= 32

You should also understand the simplest conditional statement **IF THEN ENDIF** and the slightly more complicated **IF THEN ELSE ENDIF.**

You should also be able to combine conditions in these statements using the **AND** or **OR** operators.

FUZE ®

Project stage 4-1
"Flashing lights"

In this project you will learn;

How to setup and control a simple electronic circuit;
A simple explanation of electricity
A brief introduction to resistance
Using FOR loops to control program flow

Commands & components introduced
PINMODE, DIGITALWRITE, Resistors, LEDs and Jumper wires.

Written by Jon Silvera

For educators:
Mapping with the Curriculum

Our first foray into the world of electronics is a simple but very engaging one.

Using FOR REPEAT loops we control the direction of electricity around a simple circuit to create an animated sequence of LEDs.

A deliberately layman's explanation of current flow and resistance is provided to gently introduce electronics. The real objective is to use simple electronics to further the understanding of basic programming concepts including FOR loops.

Key Stages 2, 3, 4 and beyond

Project 4 - An introduction to electronics

Here's where we start playing with very high voltage and extremely dangerous electronic equipment. You will need, rubber gloves and footwear, goggles, an earthing and anti static workstation, a dust and contamination free environment, rubber flooring and access to a 33,000 Volt AC supply, oh and a few "Danger of Death Nuclear Power" warning signs!

Just kidding - we don't need any of the above. What you will need however is a good understanding of electronics. You need to know that electricity is the transference or flow of electrons from one atom to another. Depending on the molecular structure of the material the electrons are flowing along and depending on the voltage applied along with the physical resistance imposed by any opposing force or materials then the output will vary significantly.

Erm.... Sorry, just kidding again.

Electricity is power. We use it to make things glow and move around. The electricity we use is manufactured (made) by big companies burning things like wood, coal, oil and gas. The heat they generate is converted by very clever people in white suits (scientists) into electricity and then sent all around the country to power-stations who, if we pay them lots of money, send it to our homes, offices and hospitals. We use it by plugging electrical devices into wall sockets. Some of this 'Power' is stored in batteries that we buy in shops when we need to make a toy work or a torch light up.

*Usually equal to the number of Protons, but not always the case, take Ions for example….
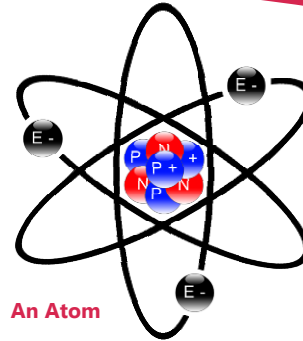
An Atom

A more complex explanation

Everything around you is made from molecules that in turn are made from different combinations of atoms.

We're talking about the very small here, from the air we breathe to the hairs on your arm, all that we know is made from atoms.

While different combinations of atoms make very different substances like water, wood, food and even smoke, atoms themselves have very similar characteristics with each other. They are all made of the same basic items, but one difference is the number of items they are made from.

An atom consists of a centre, the Nucleus, which itself holds a number of Protons (positively charged particles) and Neutrons (of no charge or neutral). Orbiting this Nucleus is a number* of negatively charged Electrons.

It is the movement or flow of these Electrons from atom to atom that defines an electric current.

To truly understand electricity, schools often employ people to teach others everything they know about it. These people are called teachers and you can easily attract their attention by jumping up and down and shouting at the top of your voice. You will want to focus your efforts on those with padded patches on their jacket elbows*. When you do capture the attention of one, feel free to instruct them to astound and fascinate you with their incredible knowledge on the subject.

So moving swiftly on, what we do need to understand is as follows;

Electricity flows around a circuit (like a racetrack). If we position things along the circuit for the electricity to flow through, it will power the item making it glow, heat up, beep or move.

Some things need more power and some less. Occasionally we need to lower the amount of power so it does not damage an item. To do this we use a **Resistor**. A resistor is a bit like treading on a hosepipe - it lets less electricity through just like your foot pressing down on the hose lets less water through.

One last thing... With our projects, as we're working with very low power, you don't need to worry too much about electric shocks. However, please do not play or experiment with any other kind of electricity, especially the kind that comes from wall sockets. Quite simply, it can hurt you, really, really hurt you. In fact, it can even kill you!

Shall we get started then?

* required due to the amount of time their elbows are spent on a desk, head in hands and groaning in pure frustration at just how difficult it is to teach basic physics & chemistry to young people these days!

Project requirements
For this project you will need the following components;

1 x 20cm Jumper cable (blue)
8 x 10cm Jumper cables (red)

8 x LED's (any colour)
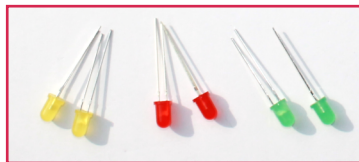
WARNING!
Connecting LED's to either of the **power** pins on the IO board without using a resistor can cause them to explode.

Unless you are using protective goggles, DO NOT DO THIS. The power pins are marked 3.3 V and 5.5 V

8 x 100R Ohms resistors colour coded
brown - **black** - **Brown** - **gold**

Firstly we need to set the 'breadboard'* up as shown in **[Pic 1]**.

You can use any colour LED and wires but in this case they are connected as follows;

One blue cable is connected to the Ground socket (GND) and another to the GPIO 0 (zero). A **Resistor** is inserted to bridge over the central gap in the breadboard. Different colour codes determine the amount of power to let through so it is important you follow the code.

LED's have an active pin which is always the longer of the two. This should be at the top as this is where our positive power supply will come from. Make sure everything is in the same line, vertically.

At this point nothing will happen because the pin sockets have not been told how to behave so;

In Direct Mode (exit the editor with **F2**) and then type;

PinMode (0,1) - press enter

This tells the computer that this pin (GPIO zero) is now configured to send out digital signals. Nothing will happen yet though as we still need to actually send a signal.

DigitalWrite (0,1) - press enter

That's better - all going well, the LED should light up **[Pic 2]** overleaf. If not check your connections thoroughly and then swap the LED.

*I would be very happy if someone could give me a good reason as to why it is called this!

Resistance is useful!

Too much electricity is never a good thing. A little LED light doesn't need much to light up, and just a bit too much can cause it to burn-out or even explode!

A **Resistor** restricts the amount of electricity allowed to pass through so the next component on the circuit is protected. We will explain resistance in greater detail at a later time. For now, just use them, Ok... good now move along please.
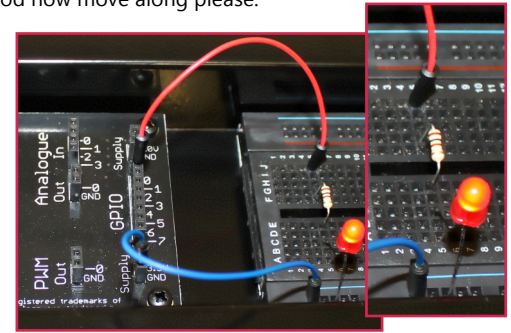
[Pic 1]

The breadboard has two lines along the top and bottom. They are are connected all the way along until the line is broken, after which another line starts again. This allows a signal to be sent all the way along with just one connection. You can then add LEDs and other components along the line to pass the signal to the main section. In this project we're connecting the blue line to the GROUND socket on the IO board using a blue cable. If we want to apply a ground signal to a new LED we can add one directly onto the blue line.

You can set any socket from 0 to 7 to **PinMode 1** (Digital Out) and then send a high signal (1) to it to send power to that pin.

Now turn it off by sending a low signal (0) to pin 0 (zero)

DigitalWrite (0,0) - press enter

And off it goes, well hopefully!

Switch to the **Editor (F2)** and start a new program **(F12)** and enter the following;

Editor	press F2 to display the editor

```
CLS
PinMode (0,1)
CYCLE
   DigitalWrite (0,1)
   Wait (.1)
   DigitalWrite (0,0)
   Wait (.1)
REPEAT
END
```

Press **F3** to **RUN**. After setting Pin 0 to Output mode we can easily send **on** (high) and **off** (low) signals to the pin to switch it on and off.

Set the breadboard up as displayed in **[Pic 3]** and update the program as shown on the next page... (**F2** to go to the Editor)

⬇

I mean, it's not like they're big enough to slice bread on, and even if you did then the crumbs would fall into the little sockets and make it pretty useless. A very silly name indeed.
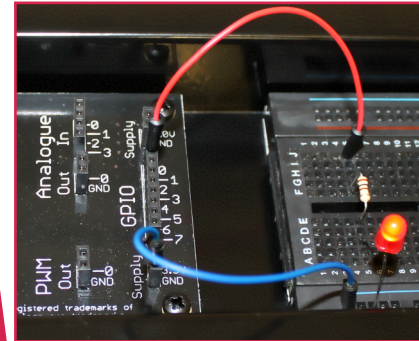
[Pic 2]

Music and lights, well lights, er well Ok... Light!

[Pic 3]
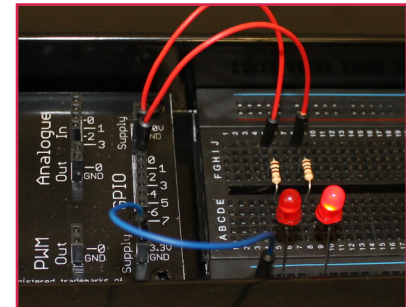
Note, we have connected **GPIO 1** to the second LED, added another resistor and connected the new LED to the **GND** signal on the blue track.

Editor	press F2 to display the editor

```
CLS
PinMode (0,1)
PinMode (1,1)
CYCLE
  DigitalWrite (0,1)
  Wait (.1)
  DigitalWrite (1,1)
  DigitalWrite (0,0)
  Wait (.1)
  DigitalWrite (1,0)
  Wait (.1)
REPEAT
END
```

Press **F3** to **RUN**. Now we have two flashing LEDs!

Set the breadboard up as shown in **[pic 4]** and then Return to the Editor and rewrite the program as shown on the next page. This will be a complete rewrite so you might as well just hit **F12** to clear the current one.

Firstly, six **GPIO** pins are configured to Digital Output mode using a simple **FOR LOOP** to set them all to 1 in succession.

Then two more **FOR LOOPS** switch the lights on and off in sequence to create an animated effect. Now turn the music up and get with the light show dudes!

So, how small is an atom really? Well, here's an interesting way to think about it…

GET WITH THE PROGRAM
Project stage 4-1
"Flashing lights"

[Pic 4]
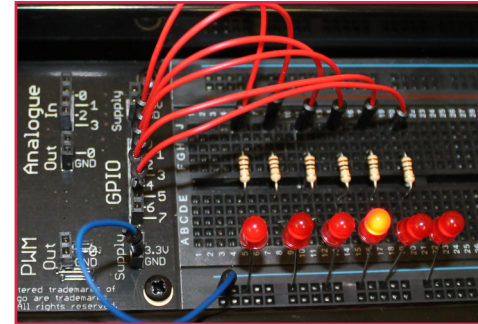
Make sure you connect the cables as follows;

GPIO 0 to the first LED (left to right)
GPIO 1 to the second LED
GPIO 2 to the third LED
GPIO 3 to the fourth LED
GPIO 4 to the fifth LED
GPIO 5 to the sixth LED

Notice we have added a resistor to every LED so there are now six LEDs, resistors and GPIO cables connected in total.

Remember the LEDs have one pin longer than the other, this is the Active pin and it needs to be at the top.

And one last time, just to be clear, nothing should be connected to the **3.3 v** or **5.5 v** power sockets.

Editor

```
CLS
FOR p = 0 TO 5 CYCLE
  PinMode (p,1)
  DigitalWrite (p,0)
REPEAT
CYCLE
  FOR p = 0 TO 5 CYCLE
    DigitalWrite (p,1)
    WAIT (0.1)
    DigitalWrite (p,0)
  REPEAT
  FOR p = 4 TO 1 STEP -1 CYCLE
    DigitalWrite (p,1)
    WAIT (0.1)
    DigitalWrite (p,0)
  REPEAT
REPEAT
END
```

A few things to take note of here;

The first **FOR** loop is used to configure the the six **GPIO** pins, 0 to 5, to digital output mode and at the same time set them all to low (off) with **DigitalWrite (p,0)**.

It's generally good practice to set things up exactly how we want them to be when we start up, so there are no surprises.

Take a small sweet, something like a Smartie and pop it in your mouth. Mmm, delicious…

The next two **FOR** loops behave very differently to each other. The first is straightforward as it simply counts through the pins, 0 to 5 one-by-one, turning each one on and then off. We've added a short, one tenth of a second delay with **WAIT (0.1)**

The second loop;

```
FOR p = 4 TO 1 STEP -1 CYCLE
```

counts in reverse (**STEP-1**) from pin 4 to pin 1. It doesn't count 0 or 5 as these are taken care of by the first loop.

Try changing it to, 5 TO 0 and see how it affects the animation. You'll notice that the first and last LED's stay on for twice as long as the rest. That's because they're switched on at the start of both loops. We want them to look the same all the way along so the second loops counts just 4 of the pins.

There's no harm in experimenting here, just make sure your program looks like it does on the left before going on to the assessments overleaf.

65

Stage 4-1 Assessments
Try the following challenges.

Assessment 1 ⭐
Increase the number of LED's to eight, and make sure they're all animated. Don't forget to increase the loop counter! You will need to add cables to the sockets numbered 6 and 7.

Assessment 2 ⭐
Adapt the program so all the LED's flash off and on at the same time. Hmm... how could you speed up the animation by so much it will look like they are all coming on at once? **Don't delay**, you'll work it out.

Assessment 3 ⭐⭐
Set up the LED's as shown and then alternate between red and green.
You'll need to count in steps and then fill in the gaps, somehow...

Assessment 4 ⭐⭐⭐
Create your own animated pattern using all three colours. You're on your own with this one. Good luck!

Assessment 1

Assessment 2

Assessment 3

Assessment 4
Rock and roll!

How many atoms are in a Smartie? There are more atoms in one single Smartie than the entire number of Smartie's ever made since they started making them in 1937. 50 Million tubes are made each year in the UK alone. In fact, you could probably add Tic-Tacs to this as well!

Project stage 4-2
"It's an Analogue World"

In this project you will learn;

How to use a Light Dependent Resistor;
The meaning of analogue and digital
The difference between
Using FOR loops to control program flow

Commands & components introduced
ANALOGREAD, ANALOGWRITE, UPDATE, INKEY, CIRCLE, TONE

Written by Colin Bodley

For educators:
Mapping with the Curriculum

Using a simple light sensor (LDR) to introduce the wonderful world of analogue is a great way to elaborate the use of variables, logic and conditional statements.

Most importantly is to develop a solid understanding that contrary to popular understanding we do not live in a digital world but an analogue one.

While it is true that we transfer and consume media digitally, our interaction with the real world is more often than not an analogue one.

Key Stages 2, 3, 4 and beyond

Analogue vs Digital

You may not be familiar with the word **Analogue** but I expect that you have heard the word **Digital** (for example when used to talk about a type of television or watch*). What do these words mean and what is the difference between them?

Analogue (or Analog in American)

Analogue values represent something in the real world such as temperature, sound or light. The values vary continuously over time.

Digital

The word digital comes from the word digit (which is another word for a finger or a single number) . Digital values can be used to represent something in the real world but they are only ever an approximate value at a given point in time.

Analogue and Digital Devices

An analogue device is something like a thermometer where the height of a column of liquid is used to represent the current temperature. A digital thermometer converts the temperature to an electrical voltage and then to a number.

* I still think that digital watches are a pretty neat idea.

Analogue vs Digital Watches

What is the difference between an analogue watch and a digital one?

An analogue watch like the one on the left uses the position of the hands to represent the time. This will vary continuously as time passes.

A digital watch like the one on the right shows a number that approximates the time at a given point.   The actual time will be somewhere between one second and the next.

Don't worry if you are finding this a bit complicated. It is enough to understand that we can measure things in the real world such as light or sound and store them in a computer as numbers.

On the other hand we can take the numbers stored in a computer and convert them back to something in the real world such as light or sound. Your digital devices such as mobile phones, MP3 players and digital TVs do this sort of thing all of the time.

You can even store the values of a sound over time and then convert it back into both light and a sound! You have probably seen programs that display patterns on a screen as music is played (called visualization).

Reading Analogue Values

Right that's enough of the boring explanation stuff, let's get on and do some programming. First you need to learn about a new FUZE BASIC command called analogueRead.

value = ANALOGREAD(inputPin)

This reads a voltage going in to one of 4 analogue inputs on the FUZE IO board and converts it to a number between 0 (0 volts) and 255 (3.3 volts).

* What did the digital watch say to the analogue watch?

Analogue to Digital Conversion

The FUZE IO Board has a special chip underneath it which can perform Analogue to Digital Conversion (and the other way around). This involves reading the voltage at a moment in time and converting it to a number (0 to 255).

So first we need something that converts our real world analogue signal and turns it into a voltage.

In the graph below the black line could represent something like the continuously changing values of a sound wave and the red line the values a particular point in time which we can store digitally in a computer.
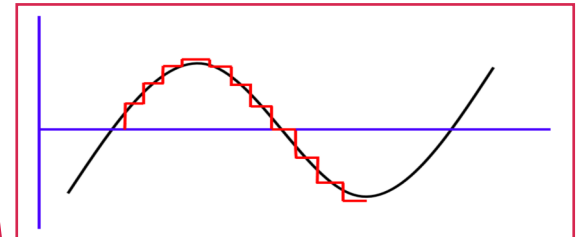
First you need to build the circuit shown on the right in **[Pic 1]**. One side of the Light Dependent Resistor is connected to the 3.3 Volt supply and the other side to Analogue In 0 on the FUZE IO Board.

Next you need to enter the following program into FUZE BASIC using the Editor as before:

Editor
| ```
CLS
CYCLE
  Pin0 = ANALOGREAD(0)
  HVTAB(10,10)
  PRINT Pin0;"    "
  UPDATE
REPEAT UNTIL Inkey = 32
END
``` |

This program loops around reading the value from Analogue In 0 and printing it on the screen until the space bar is pressed. When you **Save** and **Run** this program you should see a number printed on the screen (The **HVTAB** command makes sure that it is always printed in the same place).

The value of this number will vary depending on how dark it is where you are. You may find that it changes slightly as you are watching it.

Look no hands!

Light Dependent Resistors (LDRs)

You have already used resistors in a previous project. They reduce the amount of current flowing in an electrical circuit and at the same time reduce the voltage level. A light dependent resistor is one where the amount of resistance varies depending on how much light is shining on it. You will find one in your FUZE Electronic Components Kit:

[Pic 1]

Now try putting your hand over the top of the Light Dependent Resistor. You should see the value drop to nearly zero. If you were to shine a very bright light on to the **LDR** the value should go up to nearly the maximum (255). The more light that shines the lower the resistance and the higher the voltage is.

Now let's make this program a bit more interesting:

Editor

```
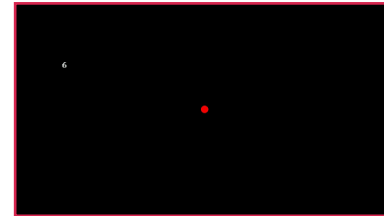COLOUR=RED
CYCLE
  CLS2
  Pin0 = ANALOGREAD(0)
  HVTAB(10,10)
  PRINT Pin0;"    "
  CIRCLE(GWIDTH/2, GHEIGHT/2, Pin0 * 2, 1)
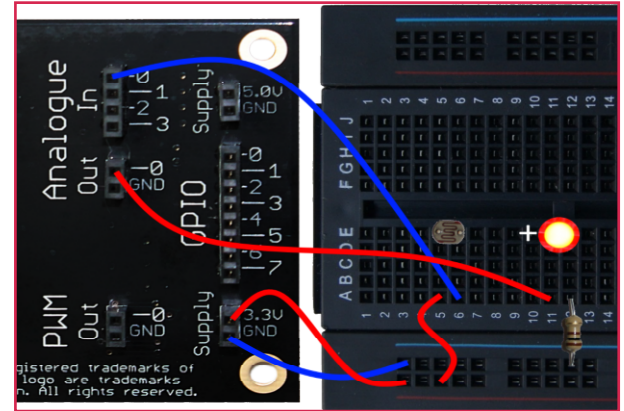  UPDATE
  WAIT(0.01)
REPEAT UNTIL Inkey = 32
END
```

This time we are using the value read from Analogue In 0 as the width of a circle drawn in the middle of the screen (actually I have doubled it to make the effect bigger). This time when you **SAVE** and **RUN** the program you will a get a red **CIRCLE** in the middle of the screen. The size of this circle will vary depending on how much light is shining on the **LDR**.

Dark

Normal Light

Bright Light

Writing Analogue Values

This still isn't very useful is it? What if we want to make a light come on gradually as it gets darker outside? Let's add a light to our original circuit. Place an **LED** (Light Emitting Diode) as shown in **[Pic 2]**. Remember that the longer leg of the **LED** is the positive (+). This should go on the left. Connect this leg to Analogue Out 0. Connect the other leg to **GND** (0 Volts) using a 100 Ohm resistor (brown, black, brown, gold).

Now we need another new command analogWrite which takes a number between 0 and 255 and converts it into a voltage between 0V and 3.3V. Change the program to add this as follows:

Editor

```
COLOUR=RED
CYCLE
  CLS2
  Pin0 = ANALOGREAD(0)
  ANALOGWRITE(0,255-Pin0)
  HVTAB(10,10)
  PRINT Pin0;"    "
  CIRCLE(GWIDTH/2,GHEIGHT/2, Pin0 * 2, 1)
  UPDATE
  WAIT(0.01)
REPEAT UNTIL Inkey = 32
END
```

Notice that we want the value to go down as the light goes up which we do by subtracting it from the maximum (255).

[Pic 2]

Digital to Analogue Conversion

This is the opposite process where we take the numbers stored in the computer and convert them back into a voltage and then to something in the real world. Because we only store the values at intervals of time the gaps in between have to be filled in. If the time interval is small enough our brains can't spot them!

Burglar Alarm

We can make a simple burglar alarm using only an **LDR** and an **LED**. Change the circuit slightly as shown in **[Pic 3]**. This will set the **LED** to permanently on. Now **CAREFULLY** bend the **LDR** down so that it is facing the **LED**. Make sure that the pins are not touching!

Finally change the program to look like this:

Editor

```
Alarm = FALSE
Threshold = 10
CYCLE
   CLS2
   Pin0 = ANALOGREAD(0)
   PRINT Pin0;"    "
   IF (Pin0 < Threshold ) THEN Alarm = TRUE
   UPDATE
   WAIT(0.1)
REPEAT UNTIL Alarm
PRINT "Alarm Activated!"
TONE(1,70,440,3)
END
```

Wh

en you run this program nothing will happen until you place a piece of paper between the **LED** and the **LDR**. Then the alarm will be activated. You may need to change the value of the *Threshold* variable depending on the amount of light where you are. The **TONE** command plays a sound but you will only hear this if your

[Pic 3]

Threshold

What do we mean by the word threshold? This is a value that when passed triggers an action to occur. In this case it is the amount of light that means that someone has passed between the **LED** and the **LDR**. You need to find a value that always sets off the alarm when someone does but doesn't go off just because it is getting dark!

Stage 4-2 Assessments

Try the following challenges:

Assessment 1 ⭐

Can you think how you might use an **LDR** and an **LED** to send a signal from one FUZE to another one? You don't have to really do it (unless you have 2 FUZEs and you want to try!)

Assessment 2 ⭐

Change the program on page 71 to make the circle get bigger as it gets darker rather than smaller. (**Hint**: see the bottom of page 72)

Assessment 3 ⭐⭐

Change the burglar alarm program so that a flashing circle is displayed on the screen when the alarm is activated. (**Hint**: Draw a red **CIRCLE**, **WAIT** and then draw a black **CIRCLE** and **WAIT** again, **REPEAT**)

Assessment 4 ⭐⭐⭐

Change the burglar alarm program and circuit so that another LED flashes when the alarm is activated (**Hint**: Project Card 4-1 Flashing Lights!)

End of Stage 4-2

Conclusion

Computers are very good at storing and processing numbers. In order to make them useful we have to turn things in the real world into these numbers. These can then be stored and even altered by a computer. Finally we can change the numbers back into things in the real world.

In the digital world of the computer everything is either 1 or 0 (True or False) but in reality these are voltages that we have assigned to these values (e.g +5V for True and 0V for False). So perhaps it is an Anlogue world after all.

GET WITH THE PROGRAM

Project stage Special 1-1

"Wir sind die Roboter"

In this project you will learn;

How to control a Robot Arm using direct programming commands;
How to send electrical signals to a circuit
How to use Procedural programming
How to create a user interface

Commands & components introduced
PROC, DEF PROC, ENDPROC, ARMBODY, ARMSHOULDER, ARMELBOW, ARMWRIST, ARMGRIPPER, ARMLIGHT, SWITCH, CASE, ENDCASE, ENDSWITCH

Written by
Jon Silvera

For educators:

Mapping with the Curriculum

If you are fortunate enough to have the FUZE Robot Arm then this is a fantastic way to introduce 'real-world' interaction and control.

By creating a simple user defined interface the user can control each joint of the Robot Arm.

The project encourages the use of Procedural programming techniques as well as complex conditional choice statements such as SWITCH and CASE.

Key Stages 2, 3, 4 and beyond

Special Stage 1-1 - The Robots

Before you attempt this project it is important to have covered some of the early steps of using **FUZE BASIC**. Please be sure to have completed Projects 1-1 and 1-2 before starting this one. Please setup your FUZE computer and connect the robot arm to one of the available USB ports. It's best to connect the Arm before running FUZE BASIC and make sure it's switched on.

What do you mean you haven't built the Robot Arm yet? Well that just won't do will it. Ok off you go then and come back once you've done it, in about three hours I reckon.

Double click the FUZE BASIC icon to begin

As you will have come to expect, FUZE BASIC will leap into action and present you with the Ready> prompt.

First of all straighten the robot arm a bit so it's not all folded up. See **[Pic 2]**. Don't worry if the arm clicks here and there this is just the gears clicking and nothing actually breaking.

Type in;

```
ArmBody (1) - press Enter
ArmBody (-1) - press Enter
ArmBody (0) - press Enter
```

Warning: These commands will set the robot arm moving as soon as you press enter. If you don't type the next command the arm will go as far as it can and start clicking - you should enter the `ArmBody (0)` command to stop it.

If at this point you get an error stating "unable to find Robot Arm" or similar then exit FUZE BASIC using `Exit` - press Enter

Ha, Robots eh… they'll never take off. Mind you they said that about flying machines once.

Robota

The first known use of the word Robot comes from the Czech "Čapek" brothers and was used in Karel Čapek's science fiction play "Rossum's Universal Robots"

[Pic 1]

The original meaning of the word Robota, was "drudgery" or "slave labour".

How awful is that!

Robots and computing

Whereas early robotic equipment was generally based on complex machines cleverly engineered to perform a specific labour saving function, it was not until the development of integrated circuits (silicon chips) and small powerful motors and electronic sensors that things really started to progress.

[Pic 2]

My favourite robot by far is "Marvin the Paranoid Android" by Douglas Adams and his comedy science fiction story The Hitchhiker's Guide to the Galaxy. Marvin is a very, very depressed robot and well known for letting us know this with quotes like; "*My capacity for happiness,*" he added, "*you could fit into a matchbox without taking out the matches first*"

Unplug the arm and reconnect it again. Also please make sure the arm is switched on **[Pic 3]**. Start **FUZE BASIC** again try the above again. If at this point it doesn't work, seek help from an adult or if you *are* an adult, from a child. On the basis things did work, try the same with these other control commands;

ArmShoulder (x) - x can be 1, -1 or 0
ArmElbow (x) - x can be 1, -1 or 0
ArmWrist (x) - x can be 1, -1 or 0
ArmGripper (x) - x can be 1, -1 or 0
ArmLight (x) - x can be 1 or 0

A useful trick to know at this point is that you can repeat the last command by pressing the up arrow key and then just edit the number. **Remember though**, you still need to press enter.

Let's put some of this new found knowledge into action. Press **F2** to enter the Editor. If there's another program listed then make sure it isn't needed and then press **F12** to clear it.

Start with the following lines of code-

Press **F3** to run the program. You will be prompted for a file name. Best to name it something like **"JSrobot"** where **JS** is your initials so you don't overwrite someone else's program.

The purpose of this section is to make sure the arm can be instructed to switch everything off, so absolutely nothing will happen at this point

Leonardo da Vinci designed and built the first known humanoid robot around 1495.

Editor
```
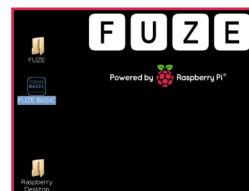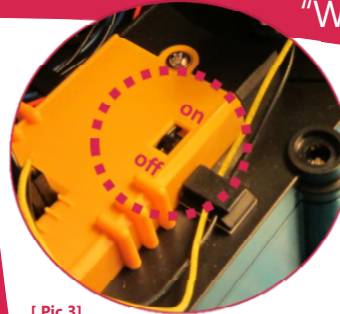CLS
PROC ResetArm
END
DEF PROC ResetArm
ARMBODY (0)
ARMSHOULDER (0)
ARMELBOW (0)
ARMWRIST (0)
ARMGRIPPER (0)
ARMLIGHT (0)
ENDPROC
```

GET WITH THE PROGRAM
Special Stage 1-1
"Wir sind die Roboter"

[Pic 3]

Today, robots can be incredibly complicated machines. Even a small toy robot can have in excess of twenty motors, light sensors, cameras, speech synthesis and significant computing power, all in a device six inches tall!

Do androids dream of electric sheep?

There are so many questions surrounding robotics but perhaps none more so than those concerned over the future of the human race. Are we developing machines that will one day become intelligent and then decide they should rule the world?

Some people believe that as mankind evolves we will enhance ourselves more and more with robotic parts and electronic circuitry. This is known as Cybernetics. The result, the Cyborg or for a simpler term, The Man Machine!

Where will it end... with the complete annihilation of the human race of course, boo!

Not to worry though as by then we will have transferred our consciousnesses into computers and so will live forever and be permanently connected to the Internet with ultra fast broadband... yay!

Edit the program to add the following - the grey text is what you should already have. You can see how it should look in **[Pic 4]**

Editor   press F2 to display the editor

```
PROC ResetArm
PROC DisplayInstructions
END
………………
DEF PROC ResetArm
ArmBody (0)
ArmShoulder (0)
ArmElbow (0)
ArmWrist (0)
ArmGripper (0)
ArmLight (0)
ENDPROC

DEF PROC DisplayInstructions
CLS
FONTSCALE (2, 2)
INK = Red
PRINT "We are the ROBOTS!"
INK = White
HVTAB (0,2)
PRINT "Press"
PRINT
PRINT "1 or 2 for Body left & right"
PRINT "3 or 4 for Shoulder up & down"
PRINT "5 or 6 for Elbow up & down"
PRINT "7 or 8 for Wrist up & down"
PRINT "9 or 0 for Gripper open & close"
PRINT "Enter to turn the Robot light on"
INK = Red
PRINT
PRINT "Space to stop movement & switch light off"
ENDPROC
```

[Pic 4]

At the moment this just displays the text as listed in the program. We still need to add the best bit - brrrr... click ..... Brr click .... Brrrrr... soon, soon, don't be so impatient!

The robot was an armoured knight that could sit up, wave its arms, and move its head while opening and closing its jaw, presumably meant to scare children who were misbehaving.
Sounds like just about every Head Teacher I've ever met!

We have introduced a couple of new commands deserving a brief explanation.

PROC DisplayInstructions and PROC ResetArm

The **PROC** command is short for Procedure. The command tells the program to jump to the part of the program labelled **DEF PROC** "**procedure name**"; in this case **DisplayInstructions** and **ResetArm**.

The end of the procedure is defined by the **ENDPROC** or End Procedure command at which point the program will return to where it was called from.

Procedures help keep a program really tidy as we can place routines and functions away from the main program. They also allow us to reuse the same routine many times with a single command. The **ResetArm** procedure for example can be used at any point to turn everything off just by using **PROC ResetArm**. It's important to grasp this as we will be using them a lot later.

FONTSCALE is very straightforward (1, 1) is normal size whereas (3, 3) is three times width and height and (2, 4) is double width but four times the height. You can experiment with this in Direct mode.

HVTAB is also very simple to grasp once explained. H is for Horiziontal and V is for Vertical. The command positions the text cursor at a specified position on the screen so that the next **PRINT** command will place the text at that position on screen. See opposite.

2040 is the year we should be in fear of, or looking forward to - you decide…

Screen coordinates
There are two ways to display information on the screen using **FUZE BASIC** or just about any other language for that matter. We either place text on the screen or graphics and pictures.

In both cases we us a simple X and Y coordinate system. When using text, the X & Y positions are determined by the width of the text so if you use huge text you will have very large grid spaces.

The example below is made possible by using the following HVTAB command;

HVTAB (1, 5)
PRINT "ROBOT"

However when we use graphics the X & Y coordinates are based on the height and width of the screen. Also the Y position 0 starts at the bottom left, not at the top as when using text.

Editor

```
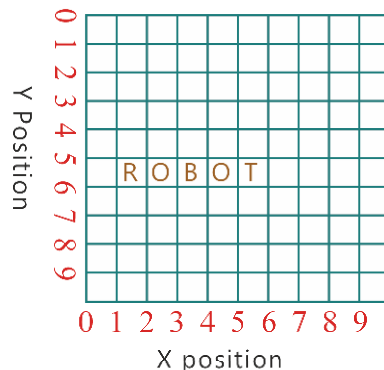PROC ResetArm
PROC DisplayInstructions

CYCLE

Key = INKEY

SWITCH (Key)
   CASE 49
    ARMBODY (1)
   ENDCASE
   CASE 50
    ARMBODY (-1)
   ENDCASE
   CASE 51
    ARMSHOULDER (1)
   ENDCASE
   CASE 52
    ARMSHOULDER (-1)
   ENDCASE
   CASE 53
    ARMELBOW (1)
   ENDCASE
   CASE 54
    ARMELBOW (-1)
   ENDCASE
   CASE 55
    ARMWRIST (1)
   ENDCASE
   CASE 56
    ARMWRIST (-1)
   ENDCASE
```

➡️

There is a lot to add here so please be sure to copy it exactly. Notice you need to continue typing the second column right after the first.

Editor

```
   CASE 57
    ARMGRIPPER (1)
   ENDCASE
   CASE 48
    ARMGRIPPER (-1)
   ENDCASE
   CASE 32
    PROC ResetArm
   ENDCASE
   CASE 13
    ARMLIGHT (1)
   ENDCASE
ENDSWITCH

REPEAT

END

DEF PROC ResetArm
ArmBody (0)
ArmShoulder (0)
ArmElbow (0)
ArmWrist (0)
ArmGripper (0)
ArmLight (0)
ENDPROC
```

More 'stuff' about robots...

The Three Laws of Robotics

The extremely renowned and very prolific science fiction author Isaac Asimov wrote hundreds upon hundreds of books, articles and short stories about science and robotics.

He introduced the idea of programming a set of rules or laws into all robots to protect humankind.

The stories written around these laws are extremely popular.

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.

2. A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.

3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Asimov later went on to add a new law to precede these - the "Zeroth Law", which focuses on humanity as a whole rather than the individual.

0. A robot may not harm humanity, or, by inaction, allow humanity to come to harm.

There are many debates as to whether we should impliment a similar set of rules into modern day robots.

Personally, I think we'll need something, otherwise who knows what might happen.

Editor

```
DEF PROC DisplayInstructions
CLS
FONTSCALE (2, 2)
INK = Red
PRINT "We are the ROBOTS!"
INK = White
HVTAB (0,2)
PRINT "Press"
PRINT
PRINT "1 or 2 for Body left & right"
PRINT "3 or 4 for Shoulder up & down"
PRINT "5 or 6 for Elbow up & down"
PRINT "7 or 8 for Wrist up & down"
PRINT "9 or 0 for Gripper open & close"
PRINT "Enter to turn the Robot light on"
INK = Red
PRINT
PRINT "Space to stop movement & switch light off"
ENDPROC
```

So two new things here; firstly the **INKEY** statement. This is a very useful command indeed. Please read the opposite page - "More about **Inkey**".

In our program we are storing the value of **Inkey** (the code value of any key pressed) in the variable **Key**

The rest is much easier than it looks. The **SWITCH** and **CASE** commands check the value stored in **Key** and depending on the value performs the command(s) in the relevant **CASE** section.

So if the "**1**" is pressed, the code value is **49** (see reference chart opposite) and so the command, **ARMBODY (1)** is executed.

It's always nice to have something to look forward to!

More about Inkey

The Inkey command is a very important one that you will use over and over again. For example, we can use Inkey to pause any program to wait (LOOP) for a key to be pressed;

> **PRINT "Press any key to continue"**
> **CYCLE**
> **REPEAT UNTIL INKEY <> -1**

If no key is being pressed the the value of **INKEY** is -1. Whenever a key is pressed its code value is stored in **INKEY**. So the above loop will repeat until Inkey is not equal to -1.

This also means we can check if a specific key is pressed. For example the value of the space bar is 32 so we could change the above to;

> **PRINT "Press the Space bar to continue"**
> **CYCLE**
> **REPEAT UNTIL INKEY = 32**

This time the program waits specifically for the space bar to be pressed and anything else will be ignored.

Here's a few more INKEY codes, just in case you need them.

48 - **0**	49 - **1**	50 - **2**	51 - **3**	52 - **4**	53 - **5**
54 - **6**	55 - **7**	56 - **8**	57 - **9**	65 - **a**	66 - **b**
67 - **c**	68 - **d**	69 - **e**	70 - **f**	71 - **g**	72 - **h**
73 - **I**	74 - **j**	75 - **k**	76 - **l**	77 - **m**	78 - **n**
79 - **o**	80 - **p**	81 - **q**	82 - **r**	83 - **s**	84 - **t**
85 - **u**	86 - **v**	87 - **w**	88 - **x**	89 - **y**	90 - **z**
32 - **Space Bar**		13 - **Enter**			

Special Stage 1-1 Assessments

Time to prove you've mastered the robot arm project.

Assessment 1 ⭐

Practice moving different parts of the robot around in Direct mode? Remember **F2** switches between Direct mode and the Editor.

Assessment 2 ⭐

Can you write a different program to display (PRINT) differerent things at different locations on the screen. Remember to Save your work before entering the **NEW** command to start a new program.

Assessment 3 ⭐⭐⭐

Write a new program to repeat a series of Robotic movements. Use the **WAIT** command to determine how far each movement goes.

*The album, The Man Machine by the German band Kraftwerk, has a very good song about Robots on it called "We are the Robots", but of course it was originally in German, hence the project title.

GET WITH THE PROGRAM
Special Stage 1-1
"Wir sind die Roboter"

Control commands
The commands are;
ArmBody (x)
ArmShoulder (x)
ArmElbow (x)
ArmWrist (x)
ArmGripper (x)
ArmLight (x)

Where x is 1, -1 or 0
The light is just 1 or 0

HVTAB (x,y)
Remember, **HVTAB** places the cursor at a set position. X is the horizontal and Y is the Vertical.

Marvin: "I am at a rough estimate thirty billion times more intelligent than you. Let me give you an example. Think of a number, any number."

Zem: "Er, five."

Marvin: "Wrong. You see?"

Marvin the paranoid android, by Douglas Adams from The Hitchhiker's Guide to the Galaxy

Appendix

For educators:
Mapping FUZE BASIC
with the Curriculum

This section aligns the FUZE BASIC projects
with various parts of the Computing
Curriculum.

Mapped by Jon Silvera

The national curriculum for computing aims to ensure that all pupils:

Can understand and apply the fundamental principles and concepts of computer science, including abstraction, logic, algorithms and data representation.

Can analyse problems in computational terms, and have repeated practical experience of writing computer programs in order to solve such problems.

Can evaluate and apply information technology, including new or unfamiliar technologies, analytically to solve problems.

Are responsible, competent, confident and creative users of information and communication technology.

The above is broken down and applied to the corresponding **Key Stages** over the following pages.

FUZE BASIC is an incredibly powerful programming language and like all versions of **BASIC** before it, is designed to be understandable to the layman and non computing specialists.

Due to its simplicity it is therefore ideal for teachers and students alike, as many core programming fundamentals can be learnt in minutes and certainly do not require pages upon pages and sites upon sites of reference material.

Additionally the programming environment is simple and self contained and as such provides instant feedback accelerating the learning process.

FUZE BASIC and the **FUZE BASIC** Projects have been specifically designed to help fulfil the requirements of the new computing curriculum. The following pages describe which projects are applicable to the various Key Stage requirements.

Additional teaching materials are available from the **FUZE** website and if you feel you have content that would fit in the same framework and would be willing to share it with the **FUZE** community then please contact admin@fuze.co.uk for the attention of Jon Silvera.

Thank you. I hope you enjoy using **FUZE BASIC** as much as we do.

Understand what algorithms are; how they are implemented as programs on digital devices; and that programs execute by following precise and unambiguous instructions.
All projects but start with 1-1 to 1-4 and Round up

Create and debug simple programs.
All projects but start with 1-1 to 1-4 and Round up

Use logical reasoning to predict the behaviour of simple programs.
All projects but start with 1-1 to 1-4 and Round up

Use technology purposefully to create, organise, store, manipulate and retrieve digital content.
Projects 1-2, 1-3

Recognise common uses of information technology beyond school.
Projects 1-4, 4-1, 4-2 and Special Stage 1-1

Use technology safely and respectfully, keeping personal information private; identify where to go for help and support when they have concerns about content or contact on the Internet or other online technologies.
Not directly relevant to programming

Design, write and debug programs that accomplish specific goals, including controlling or simulating physical systems; solve problems by decomposing them into smaller parts.
All projects but start with 1-1 to 1-4 and Round up. Note control and physical systems are included in 4-1, 4-2 and SS 1-1

Use sequence, selection, and repetition in programs; work with variables and various forms of input and output.
Projects 1-2, 1-3, Round Up, 2-1

Use logical reasoning to explain how some simple algorithms work and to detect and correct errors in algorithms and programs.
All projects but start with 1-1 to 1-4 and Round up

Understand computer networks including the Internet; how they can provide multiple services, such as the world wide web; and the opportunities they offer for communication and collaboration.
Not included at this time but further Projects will include networking, TCP/IP, Internet and other protocols.

Use search technologies effectively, appreciate how results are selected and ranked, and be discerning in evaluating digital content.
Not directly relevant to programming

Select, use and combine a variety of software (including Internet services) on a range of digital devices to design and create a range of programs, systems and content that accomplish given goals, including collecting, analysing, evaluating and presenting data and information.
All projects

Use technology safely, respectfully and responsibly; recognise acceptable/unacceptable behaviour; identify a range of ways to report concerns about content and contact.
Not directly relevant to programming

Design, use and evaluate computational abstractions that model the state and behaviour of real-world problems and physical systems.
All projects but start with 1-1 to 1-4 and Round up. Note control and physical systems are included in 4-1, 4-2 and SS 1-1

Understand several key algorithms that reflect computational thinking [for example, ones for sorting and searching]; use logical reasoning to compare the utility of alternative algorithms for the same problem.
At this time this is not comprehensively covered but these projects do offer an insight; 1-2, 1-3, Round Up and 2-1

Use two or more programming languages, at least one of which is textual, to solve a variety of computational problems; make appropriate use of data structures [for example, lists, tables or arrays]; design and develop modular programs that use procedures or functions.
FUZE BASIC is a text based language. All projects.

Understand simple Boolean logic [for example, AND, OR and NOT] and some of its uses in circuits and programming; understand how numbers can be represented in binary, and be able to carry out simple operations on binary numbers [for example, binary addition, and conversion between binary and decimal].
Projects 1-4, 2-1 and 4-2

Understand the hardware and software components that make up computer systems, and how they communicate with one another and with other systems.
Not directly relevant to programming but there are a number of projects that communicate with external devices. These include 4-1, 4-2 and SS 1-1

Understand how instructions are stored and executed within a computer system; understand how data of various types (including text, sounds and pictures) can be represented and manipulated digitally, in the form of binary digits.
Projects 1-2, 1-3, 1-4, Round Up and 2-1

Undertake creative projects that involve selecting, using, and combining multiple applications, preferably across a range of devices, to achieve challenging goals, including collecting and analysing data and meeting the needs of known users.
Projects 1-2, 1-3, 1-4, Round Up and 2-1

Create, re-use, revise and re-purpose digital artefacts for a given audience, with attention to trustworthiness, design and usability.
Not covered

Understand a range of ways to use technology safely, respectfully, responsibly and securely, including protecting their online identity and privacy; recognise inappropriate content, contact and conduct and know how to report concerns.
Not directly relevant to programming

All pupils should be taught to: develop their capability, creativity and knowledge in computer science, digital media and information technology.
All projects

Develop and apply their analytic, problem-solving, design, and computational thinking skills.
All projects

Understand how changes in technology affect safety, including new ways to protect their online privacy and identity, and how to identify and report a range of concerns.
Not directly relevant to programming

FUZE®

Get with the PROGRAM

Powered by Raspberry Pi

fuze.co.uk